

# **Introduction to Direct3D 10**

**SIGGRAPH 2007 Course 5**

**August 5, 2007**

**Course Organizer:**

**Chuck Walbourn, Microsoft Corp**

**Lecturers:**

**Daniel Barrero, Relic Entertainment**

**Chas Boyd, Microsoft Corp**

**Sam Glassenberg, Microsoft Corp**

**Michael Oneppo, Microsoft Corp**

**Nick Porcino, LucasArts**

**Doug Service, Microsoft Game Studios**

**Chuck Walbourn, Microsoft Corp**

**Carsten Wenzel, Crytek**

## About This Course

The general release of Windows Vista in January 2007 included a major new revision of the Direct3D graphics platform. Unlike previous generations of Direct3D, version 10 is a radical redesign of the entire rendering stack in hardware and software. The Direct3D 10 API has been designed to significantly streamline the rendering pipeline for greatly increased performance, stability, consistency, and greatly simplified configuration for applications while providing increased generality and flexibility. While there are many new opportunities, there are also new challenges with porting existing graphics code, extending support across a broad range of platforms, and creating scalable and exploitative content.

This course equips the attendee with a broad understanding of the new technology, detailed insight into the rendering pipeline and shading language extensions, a working knowledge of getting up and running on Direct3D 10, and practical advice for creating, porting, and tuning Direct3D 10 applications.

## Prerequisites

This course assumes attendees are familiar with developing Direct3D 9 or OpenGL graphics applications, and have a basic proficiency in writing code in a high-level programmable shader language such as HLSL, Cg, or GSL.

## Topics

*Bootstrapping Direct3D 10* (Chuck Walbourn): A high-level system overview of Direct3D 10 and how it relates to Direct3D 9 and other graphics APIs. We will run through the basics of setting up and creating a Direct3D 10 device, and demonstrate a few Direct3D 10 code/shader equivalents for existing fixed-function style computations.

*The Direct3D 10 Pipeline* (Chas Boyd): A tour through the Direct3D 10 rendering pipeline with a detailed review of the APIs related to each stage. Comparisons with existing graphics APIs are called out to help bring the audience up to speed with the new design.

*HLSL Shader Model 4.0* (Michael Oneppo): Learn about advancements in the HLSL language to support the more general and robust programming model in Shader Model 4.0. This talk covers use of geometry shaders, integer instructions, new texture intrinsics and flow control mechanisms. Techniques for developing shader content to be shared between Direct3D9 and 10 are also discussed.

*Effects 10* (Sam Glassenberg): A review the Direct3D 10 Effects System – a series of APIs to efficiently abstract and manage GPU device state, shaders, and constants. This talk covers the methods to reflect and manage material content as effect (.fx) files.

*Porting Game Engines to Direct3D 10:* Direct3D 10 is a major revision of the Direct3D API, and poses unique challenges to games supporting multiple platforms while still taking advantage of the new capabilities. This presentation covers practical experiences in updating game engine technology to support and exploit Direct3D 10, and the implications for hosting both Direct3D 9 and Direct3D 10 within the same executable.

*Crytek's Crysis* (Carsten Wenzel)

*Relic Entertainment's Company of Heroes* (Daniel Barrero)

*Microsoft's Flight Simulator X* (Doug Service)

*Content Tools and Film Use of Direct3D 10* (Nick Porcino): The Direct3D 10 pipeline offers unique opportunities for producing high-end graphical content, but requires a new generation of tool capabilities. With the new leap in consumer-level Direct3D hardware capabilities, interest in film pre-visualization and GPU-based acceleration continues to grow.

*Debugging Direct3D 10 Applications* (Chuck Walbourn): The new Direct3D 10 technology provides a series of new debugging and profiling mechanisms for graphics code, and leverages tools like *PIX for Windows* to provide shader-debugging functionality. This topic covers the use of these facilities and demonstrates how to use the new tools for debugging Direct3D 10 applications.

*Performance Tuning for Direct3D 10* (Chas Boyd): The new Direct3D 10 architecture is designed to enable applications to minimize the CPU overhead of rendering. This session reviews design strategies and best-practices to maximize performance on the new API and hardware. This topic reviews the standard usage idioms expected for Direct3D 10, with comparisons to existing Direct3D 9 code patterns.

## **Recommended Reading**

DirectX SDK Documentation - Direct3D 10

<http://msdn2.microsoft.com/en-us/library/bb205066.aspx>

DirectX SDK Technical Article – Graphics APIs in Windows Vista

<http://msdn2.microsoft.com/en-us/library/bb173477.aspx>

## **Acknowledgements**

Thanks to our speakers: Daniel Barrero, Chas Boyd, Sam Glassenberg, Michael Oneppo, Nick Porcino, Doug Service, and Carsten Wenzel.

Their companies: Crytek, LucasArts, Microsoft - ACES, Microsoft - Direct3D, Microsoft - XNA Developer Connection, and Relic Entertainment.

Thanks to Shanon Drone, Matt Dudley, and Cyrus Kanga for the course image.

Special thanks to Kris Gray and Jeremy Gup for their help in preparing these course notes.

## Lecturers

### **Daniel Barrero, Graphics Programmer, Relic Entertainment**

Daniel Barrero is currently a graphics programmer at Relic Entertainment Canada where he has worked on Warhammer 40,000:Winter Assault, The Outfit and Company of Heroes. He received Computer Engineering Bachelor's and Master's degrees from the University of Los Andes in Bogota, Colombia, an advanced studies degree on computer graphics (D.E.A.) and a PhD in Computer Science at the IRIT-University Paul Sabatier Toulouse III in France. Previously to working on the video game industry he was a Postdoctoral fellow at the mechanical engineering department of the Polytechnic School of Montreal conducting research on Scientific Visualization, Fluid Dynamics, and Virtual Reality.

### **Chas Boyd, Software Architect, Microsoft Corporation**

Chas Boyd joined the Direct3D team in 1995 and has contributed to releases since DirectX 5. Over that time he worked closely with hardware and software developers to drive the adoption of features like programmable hardware shaders and floating-point pixel processing. He has developed and demonstrated initial hardware accelerated versions of techniques including hardware soft skinning, hemispheric lighting with ambient occlusion, matrix palette skinning, and N-Patches. Earlier he worked in the areas of scientific visualization and 3D modeling, and more recently on games and on APIs for GPGPU computing.

### **Sam Glassenberg, Lead Program Manager – Direct3D, Microsoft Corporation**

Sam started at Microsoft in 2002. He currently manages the Direct3D Program Management team, delivering a series of graphics technologies including the Direct3D graphics APIs and shader technologies. His prior experience includes animation work at LucasArts, as well as contributions to a series of medical and surgical simulation projects. Sam received his Masters in Computer Science from Stanford University with a focus on computer graphics.

### **Michael Oneppo, Program Manager – Direct3D, Microsoft Corporation**

Michael Oneppo is a program manager on the Microsoft Direct3D team, currently focusing on D3DX and shader technologies for Direct3D 10 and beyond. Michael received his Sc. B. in Computer Science from Brown University, focusing on graphics for VR and visualization.

### **Nick Porcino, Senior Software Engineer, LucasArts**

When Nick Porcino saw Star Wars as a kid back in 1977, and Jim Blinn's Voyager flyby animation a few years later, he set his sights on applying computer graphics and simulation technologies to entertainment. He published his first computer game in 1981, his first console game in 1984, and his first autonomous submersible in 1992. Today he works with artists and engineers at LucasArts and Industrial Light + Magic to create and converge technologies at the interface between games and film.

**Doug Service, Software Development Engineer, Microsoft Game Studios**

Doug Service is a member of the Flight Simulator development team at Microsoft's ACES Game Studio. His primary responsibility is the design and implementation of rendering systems that use the D3D10 API. Doug is an avid sailplane pilot and working on Flight Simulator is a labor of love. Previously, Doug was the Director of Technology Development at Ritual Entertainment where he led the design and implementation of Ritual's next generation game engine. He taught math and physics for game developers as an adjunct professor at Southern Methodist University's Guildhall program.

**Chuck Walbourn, Software Design Engineer, Microsoft Corporation**

Chuck has been working on games for the Windows platform since the early days of DirectX and *Windows 95*. He got his start in the game industry starting his own development house during the mid-90s in Austin, shipped several Windows titles for *Interactive Magic* and *Electronic Arts*, and developer the content tools pipeline for the *Microsoft Game Studios* Xbox title *Voodoo Vince*. He currently works in the XNA Developer Connection group at Microsoft supporting and educating game developers working on the Windows platform. Chuck holds a Bachelors and Masters of Computer Science from the University of Texas at Austin.

**Carsten Wenzel, Software Engineer, Crytek**

Carsten Wenzel is a software engineer and member of the R&D staff at Crytek. During the development of FAR CRY he was responsible for performance optimizations on the CryEngine. Currently he is busy working on the next iteration of the engine to keep pushing future PC and next-gen console technology. Prior to joining Crytek he received his M.S. in Computer Science at Ilmenau, University of Technology, Germany in early 2003.

## **Course Notes**

Updated course notes are available at: <http://msdn.com/directx/presentations>

## **Resources**

DirectX Developer Center

<http://msdn.microsoft.com/directx/>

DirectX Software Development Kit (SDK)

<http://msdn2.microsoft.com/en-us/xna/aa937788.aspx>

Direct3D 10 Tutorials and Samples

<http://msdn2.microsoft.com/en-us/library/bb205282.aspx>

AMD/ATI Developer Site

<http://ati.amd.com/developer/>

NVidia Developer Site

<http://developer.nvidia.com/>

## Table of Contents

Course Introduction	1
Bootstrapping Direct3D 10	11
The Direct3D 10 Pipeline	52
HLSL Shader Model 4.0	111
Effects 10	153
Porting Game Engines to Direct3D 10 - Crytek's <i>Crysis</i>	207
Porting Game Engines to Direct3D 10 - Relic Entertainment's <i>Company of Heroes</i>	223
Porting Game Engines to Direct3D 10 - Microsoft's <i>Flight Simulator X</i>	250
Content Tools and Film Use of Direct3D 10	275
Debugging Direct3D 10 Applications	299
Performance Tuning for Direct3D 10	322
Direct3D 10 Programming Guide Excerpts	369



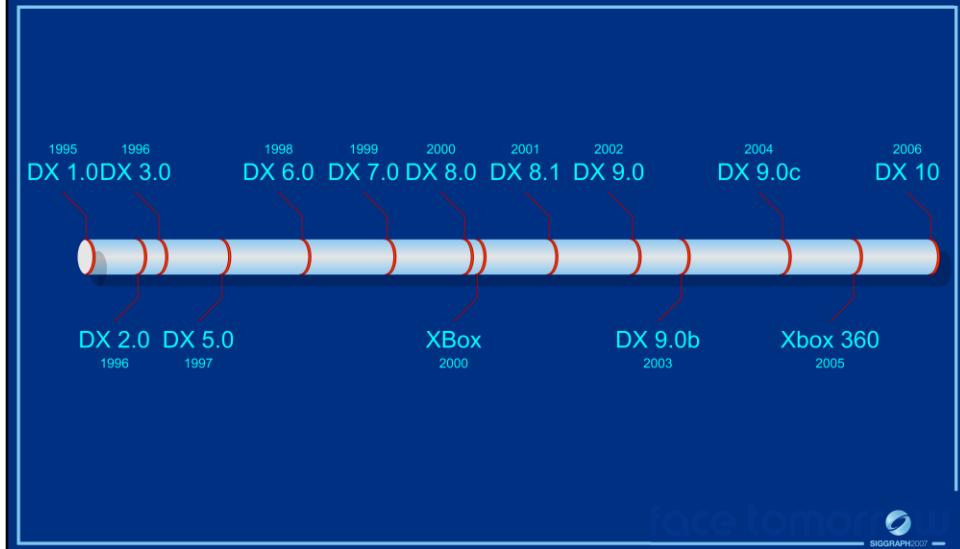
**SIGGRAPH2007**

# Introduction to Direct3D 10 Course

Chuck Walbourn  
Microsoft Corp



# Evolution of Direct3D



Direct3D 2 / 3 (1995/6) - Software rendering and first-generation PC consumer 3D hardware

Direct3D 5 (1997) - DrawPrimitive API

Direct3D 6 (1998) - DXTn compression, bump mapping, SSE & 3DNow! optimization

Direct3D 7 (1999) - Fixed-func hardware Transform and Lighting

Direct3D 8 (2000) - Vertex & Pixel Shaders (version 1.0 / 1.1)

Direct3D 8.1 (2001) - Pixel Shader 1.x, HLSL; Xbox

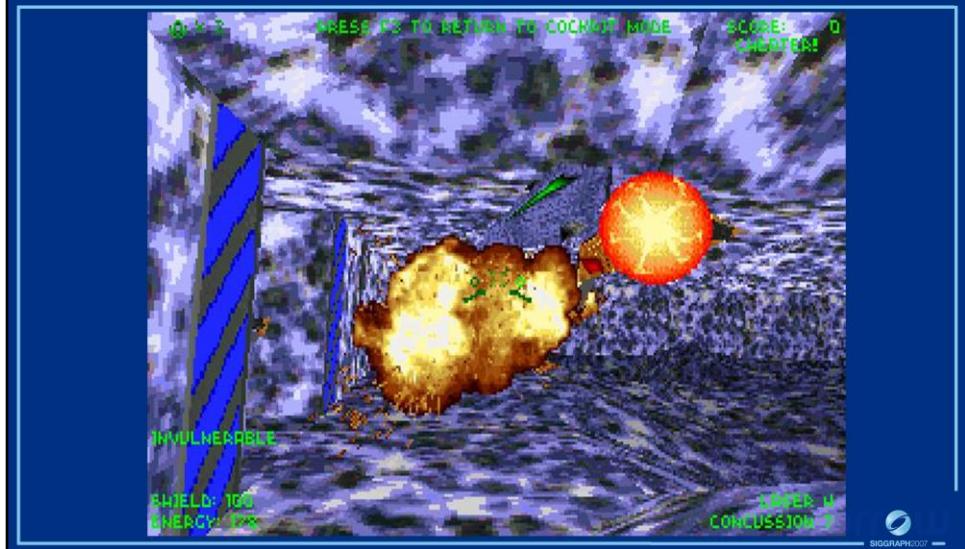
Direct3D 9 (2002) - Shader Model 2.0, MRTs, VB indexing

Direct3D 9.0b (2003) - Shader Model 2.x

Direct3D 9.0c (2004) - Shader Model 3.0, h/w instancing; Xbox 360

Direct3D 10 (2006)

## *Descent*, id Software (1995)



## **Half-Life 2, Valve Software (2004)**



## *Crysis*, Crytek



SIGGRAPH2007

# Direct3D 10 Design Goals

- Performance
  - Greatly improved 'small batch' performance, reduce validation overhead, in-GPU multipass (Stream Out)
- Consistency
  - Eliminate 'capability bits', unify shader cores, strictly defined behavior across chipsets
- Visual Quality
  - More expressive programmability, geometry shader, texture compression formats for normals, resource views



## Revolution vs Evolution

- Direct3D 9 now a mature API
- To address goals required fundamental architectural change
  - Completely rewritten API software stack
- Direct3D 10 leverages new Windows Display Driver Model introduced in Windows Vista
  - Completely rewritten driver hardware stack

face tomorrow  SIGGRAPH 2007

## The Fate of Direct3D 9

- Direct3D 9 is now a core part of Windows
  - Replaces GDI as the primary graphics API
  - More closely matches modern video hardware
  - Now focused on the needs of ‘mainstream’ applications
- Direct3D 10 and future versions will continue to focus on high-end graphics applications

face tomorrow



“Graphics APIs in Windows Vista”

<<http://msdn2.microsoft.com/en-us/library/bb173477.aspx>>

## Course Schedule

- |   |  |
|---|--|
| <ul style="list-style-type: none"><li>• 8:35 – 9:15a</li><li>• 9:15 – 10:15a</li><li>• 10:30 – 11:30a</li><li>• 11:30 – 12:15p</li><li>• 1:45 – 3:00p</li><li>• 3:00 – 3:30p</li><li>• 3:45 – 4:15p</li><li>• 4:15 – 5:15p</li><li>• 5:15 – 5:30p</li></ul> | <ul style="list-style-type: none"><li>• Bootstrapping Direct3D 10</li><li>• The Direct3D 10 Pipeline</li><li>• HLSL Shader Model 4.0</li><li>• Effects 10</li><li>• Porting Game Engines to Direct3D 10</li><li>• Content Tools and Film Use of D3D 10</li><li>• Debugging Direct3D 10 Applications</li><li>• Performance Tuning for Direct3D 10</li><li>• Q/A</li></ul> |
|---|--|

face tomorrow



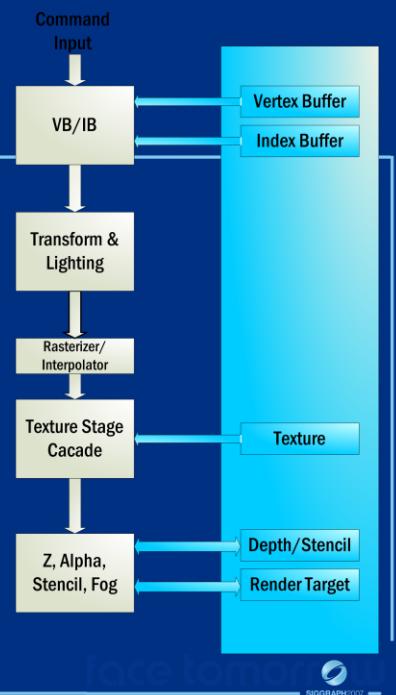




*Bootstrapping Direct3D 10* (Chuck Walbourn): A high-level system overview of Direct3D 10 and how it relates to Direct3D 9 and other graphics APIs. We will run through the basics of setting up and creating a Direct3D 10 device, and demonstrate a few Direct3D 10 code/shader equivalents for existing fixed-function style computations. [40 minutes]

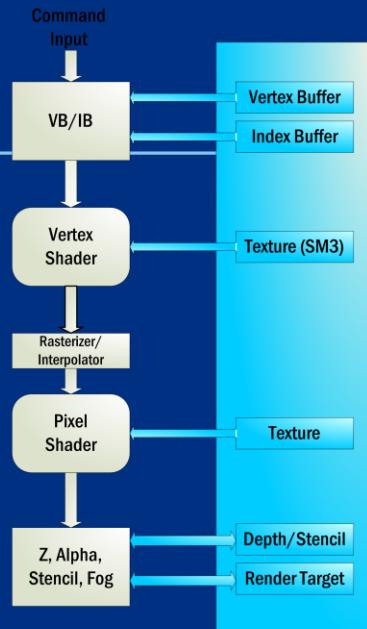
## Direct3D 9 Pipeline (Fixed-Function)

- Similar to Direct3D 7
- Fixed-function  $W^*V^*P$  transformation, point/spot/ambient lighting, and multi-texture blending cascade



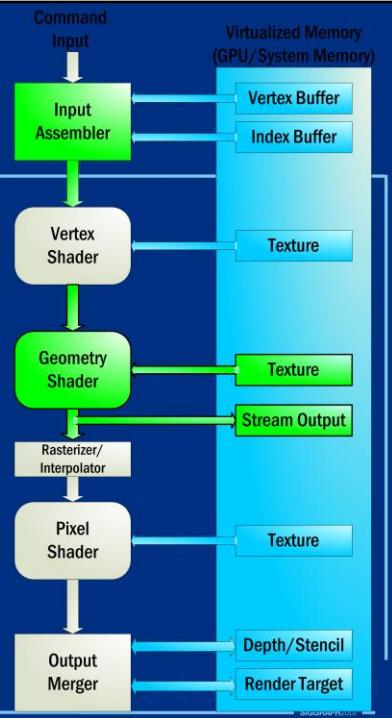
## Direct3D 9 Pipeline (Shaders)

- Originally introduced with Direct3D 8
- Could be mixed with Fixed-Function pipeline by using FVF's and "NULL" shader programs



# Direct3D 10 Pipeline

- Formalized fetch from VB/IB as Input Assembler
- Formalized stencil/z-test and alpha-blend as Output Merger
- New stages: Geometry Shader, Stream Out
- Eliminates legacy fixed-function T&L/TSS



## Major changes from Direct3D 9

- Common shader core
  - Vertex and Pixel shaders have the same instruction set, access to resources, etc.
- Guaranteed Feature Set
  - No 'caps bits' and strictly defined hardware rules
- Programmable shader model
  - All rendering requires the use of shaders
  - Shaders must be authored in HLSL SM 4.0



## Major changes from Direct3D 9

- Resources now two logical pieces
  - Buffer to contain the actual bits
  - A view to control interpretation of those bits and binding to the pipeline stages
  - Multiple views can reference the same buffer
- GPU and Video Memory now virtualized
  - No more “LOST DEVICE” condition
  - Multiple Direct3D applications can now properly multitask and share the GPU

face tomorrow



## Major changes from Direct3D 9

- Completely redesigned API
  - More closely reflects underlying hardware
  - Isolates state/resource validation to specific times, preferably at creation

face tomorrow  SIGGRAPH 2007

## Direct3D 9 v 10 API

IDirect3DDevice9

ID3D10Device

Many method names  
prefixed to reflect pipeline  
stage:

IA = Input Assembler

VS = Vertex Shader

GS = Geometry Shader

SO = Stream Out

RS = Rasterizer Stage

PS = Pixel Shader

OM = Output Merger



## Direct3D 9 v 10 API

IDirect3D9

IDXGIFactory,  
IDXGIAdapter,  
IDXGIDevice

IDirect3DDevice9

IDXGISwapChain

::Present

::Present

IDirect3DDevice9

IDXGISwapChain

::TestCooperativeLevel

::Present with  
PRESENT\_TEST

Take tomorrow's  
SIGGRAPH 2007



## DirectX Graphics Infrastructure (DXGI)

- Each generation of Direct3D has had code for enumeration, device creation, swap chains, etc.
- The DXGI Library on Windows Vista takes over this responsibility
  - Future versions of Direct3D will also use DXGI
  - DXGI interfaces documented in the DirectX SDK
  - `D3DFORMAT` type now `DXGI_FORMAT` type

Take tomorrow's  
SIGGRAPH 2007



# DXGI Library

## IDXGIAdapter

- Video Card
  - Description, video memory size, etc.
  - Enumerates IDXGIOutput



## IDXGIOutput

- Monitor
  - Mode list, frame statistics, gamma control



SIGGRAPH 2007

# DXGI Swap Chain

IDXGISwapChain

::SetFullScreenState

Switch between  
windowed and  
fullscreen mode

::ResizeTarget

Changes display mode in  
fullscreen or window  
size

::ResizeBuffers

Resizes back buffers



## Direct3D 9 v 10 API

IDirect3DBaseTexture9	ID3D10Buffer
IDirect3DTexture9	ID3D10Texture1D
IDirect3DCubeTexture9	ID3D10Texture2D
IDirect3DVolumeTexture9	ID3D10Texture3D
IDirect3DIndexBuffer9	
IDirect3DVertexBuffer9	



Most resources are ID3D10Buffers from ID3D10Device::CreateBuffer() and are given bind types of D3D10\_BIND\_VERTEX\_BUFFER , D3D10\_BIND\_INDEX\_BUFFER, D3D10\_BIND\_CONSTANT\_BUFFER, or D3D10\_BIND\_STREAM\_OUTPUT

Texture resources are created as ID3D10Texture(1-3)D resources created by ID3D10::CreateTexture(1-3)D and given the bind types of D3D10\_BIND\_SHADER\_RESOURCE, D3D10\_BIND\_RENDER\_TARGET, or D3D10\_BIND\_DEPTH\_STENCIL. These can be arrays.

## Direct3D 9 v 10 API

IDirect3DVertexShader9      ID3D10VertexShader

IDirect3DPixelShader9      ID3D10PixelShader

New shader stage      ID3D10GeometryShader

IDirect3DVertexDeclaration9      ID3D10InputLayout

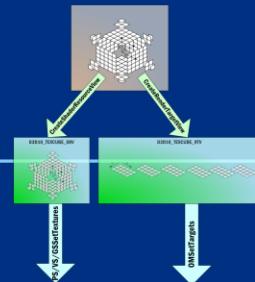
face tomorrow



Shaders are created via `ID3D10Device::CreateVertexShader()`,  
`::CreateGeometryShader()`, or `::CreatePixelShader()`.

Input layouts are created by `ID3D10Device::CreateInputLayout()`

## Direct3D 9 v 10 API



New concept: Views  
vs. Buffers

Splits resource data  
from type

Allows the same data  
to be reinterpreted  
as multiple type

ID3D10ShaderResourceView

For textures and raw buffer views  
bound to VS, GS, or PS

ID3D10RenderTargetView

ID3D10DepthStencilView

For binding to OM

Views are created via `ID3D10Device::CreateShaderResourceView()`,  
`::CreateRenderTargetView()`, or `::CreateDepthStencilView()`.

Shader resources are bound to shader stages the pipeline via  
`ID3D10Device::(V,G,P)SSetShaderResources()`.

Note that Constant Buffers are bound to shader stages via  
`ID3D10Device::(V,G,P)SSetConstantBuffers()`, not through the use of views.

## Direct3D 9 v 10 API

IDirect3DDevice9

::SetRenderState

ID3D10BlendState

ID3D10DepthStencilState

ID3D10RasterizerState

IDirect3DDevice9

::SetTextureStageState

ID3D10SamplerState

Fixed-function T&L and  
TSS states removed



State objects are created via ID3D10Device::CreateBlendState(), ::CreateDepthStencilState(), ::CreateRasterizerState(), and ::CreateSamplerState(). Once created, state objects are immutable.

Blend and DepthStencil state are bound to the OM stage via ID3D10Device::OMSetBlendState() and ::OMSetDepthStencilState().

Rasterizer state is bound to the RS stage via ID3D10Device::RSSetState().

Sampler stage is bound to each shader stage of the pipeline via ID3D10::(V,G,P)SSetSamplers().

## Direct3D 9 v 10 API

<b>IDirect3DDevice9</b>	<b>ID3D10Device</b>
::DrawIndexedPrimitive	::Draw
::DrawPrimitive	::DrawIndexed
	::DrawIndexedInstanced
	::DrawInstanced
	::IASetPrimitiveTopology
<b>New for Stream Out stage</b>	::DrawAuto

face tomorrow SIGGRAPH2007

The primitive topology on Direct3D 9 was given as part of the draw call itself: POINTLIST, LINELIST, LINESTRIP, TRIANGELIST, TRIANGLESTRIP, TRIANGELFAN.

On Direct3D 10, the primitive topology is a state set by ID3D10Device::IASetPrimitiveTopology(). Supports POINTLIST, LINELIST, LINESTRIP, TRIANGELIST, TRIANGLESTRIP, LINELIST\_ADJ, LINESTRIP\_ADJ, TRIANGELIST\_ADJ, and TRIANGLESTRIP\_ADJ.

Note that TRIANGLEFAN is no longer supported as it was seldom used and is easy to convert to a LIST or STRIP.

## Direct3D 9 v 10 API

IDirect3DDevice9

Removed

::BeginScene

::EndScene

::DrawPrimitiveUP

::DrawIndexedPrimitiveUP

face tomorrow



## Direct3D 9 v 10 API

### IDirect3DDevice9

::ShowCursor  
::SetCursorPosition  
::SetCursorProperties

Removed as standard  
Win32 mouse cursors  
work with Direct3D

::Reset

No more LOST device  
case

face tomorrow 

## Direct3D 9 v 10 API

IDirect3DDevice9

Fixed-function removed

- ::DrawRectPatch
- ::DrawTriPatch
- ::LightEnable
- ::MultiplyTransform
- ::SetLight
- ::SetMaterial
- ::SetNPatchMode
- ::SetTransform
- ::SetFVF

face tomorrow



## Direct3D 9 v 10 API

IDirect3DDevice9  
    ::CheckDepthStencilMatch  
    ::CheckDeviceFormat  
    ::GetDeviceCaps  
    ::ValidateDevice

“Capability bits” removed

Only a few format usage  
cases are optional and  
checked with

ID3D10Device  
    ::CheckFormatSupport

face tomorrow



## Starting up a Direct3D 9 Device

```
g_pD3D = Direct3DCreate9( D3D_SDK_VERSION ) ;

if (!g_pD3D) return E_FAIL;

D3DPRESENT_PARAMETERS pp;
memset(&pp, 0, sizeof(pp));
pp.Windowed = TRUE;
pp.SwapEffect = D3DSWAPEFFECT_DISCARD;
pp.BackBufferFormat = D3DFMT_UNKNOWN;

HRESULT res = d_pD3D->CreateDevice( D3DADAPTER_DEFAULT,
    D3DDEVTYPE_HAL, hWnd, D3DCREATE_HARDWARE_VERTEXPROCESSING, &pp,
    &g_pDevice );
if ( FAILED(res) ) // Error handling
```

SIGGRAPH2007

# Direct3D 9: Render Loop

```
g_pDevice->Clear( 0, NULL, D3DCLEAR_TARGET, D3DCOLOR_XRGB(0,0,0),  
    1.f, 0 );  
  
g_pDevice->BeginScene();  
// Do render  
g_pDevice->EndScene();  
  
res = g_pDevice->Present( NULL, NULL, NULL, NULL );  
if ( FAILED(res) )  
{  
    // Handle D3DERR_DEVICELOST and other errors  
}
```

SIGGRAPH2007

# Starting up a Direct3D 10 Device

```
DXGI_SWAP_CHAIN_DESC sd;  
memset(&sd, 0, sizeof(sd));  
sd.BackBufferCount = 1;  
sd.BackBuferDesc.Width = 640;  
sd.BackBufferDesc.Height = 480;  
sd.BackBufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;  
sd.BackBufferDesc.RefreshRate.Numerator = 60;  
sd.BackBufferDesc.RefreshRate.Demoninator = 1;  
sd.OutputWindow = hWnd;  
sd.SampleDesc.Count = 1;  
sd.Windowed = TRUE;  
...  
...
```

SIGGRAPH2007

## Starting up a Direct3D 10 Device

```
...  
HRESULT res = D3D10CreateDeviceAndSwapChain( NULL,  
    D3D10_DRIVER_TYPE_HARDWARE, NULL, 0, D3D10_SDK_VERSION, &sd,  
    &g_pSwapChain, &g_pDevice );  
  
if ( FAILED(res) ) // Error Handling
```

SIGGRAPH2007

## Direct3D 10: Bind render target

```
// Bind swap-chain as render target view  
  
ID3D10Texture2D *bb;  
  
res = g_pSwapChain->GetBuffer( 0, __uuidof(ID3D10Texture2D) ,  
    (LPVOID*) &bb );  
  
if ( FAILED(res) ) // Error handling  
  
  
res = g_pDevice->CreateRenderTargetView( bb, NULL, &g_pRTV );  
bb->Release();  
  
if ( FAILED(res) ) // Error handling  
  
  
g_pDevice->OMSetRenderTargets(1, &g_pRTV, NULL);
```

SIGGRAPH2007

## Direct3D 10: Initialize viewport

```
// Set initial viewport  
  
D3D10_VIEWPORT vp;  
  
vp.Width = 640.f;  
vp.Height = 480.f;  
vp.MinDepth = 0.f;  
vp.MaxDepth = 1.f;  
vp.TopLeftX = 0.f;  
vp.TopLeftY = 0.f;  
  
g_pDevice->RSSetViewports( 1, &vp );
```

SIGGRAPH2007

## Direct3D 10: Render Loop

```
// Clear backbuffer
float color[4] = { 0.f, 0.f, 0.f, 1.f };

g_pDevice->ClearRenderTargetView( g_pRTV, color );

// Do render

res = g_pSwapChain->Present( NULL, NULL, NULL, 0, 0, 0 );

if ( FAILED(res) )

{

    // Handle rare error condition DXGI_ERROR_DEVICE_REMOVED
    // and other errors

}

else if (res == DXGI_STATUS_OCCLUDED) // Minimized or hidden
```

SIGGRAPH2007

## Aside: Driver types

- **D3D10\_DRIVER\_TYPE\_HARDWARE**
  - Full-speed hardware implementation
  - Requires Direct3D 10 class (or later) hardware running on Windows Vista
- **D3D10\_DRIVER\_TYPE\_REFERENCE**
  - Software renderer for compliance testing, not performance
  - Requires Windows Vista and DirectX SDK
- **D3D10\_DRIVER\_TYPE\_NULL**
  - Reference device with no support for rendering
  - Requires Windows Vista

# Programmable Shaders

- Vertex Shaders
  - Transformation, lighting
  - Per-vertex operations like skinning
- Pixel Shaders
  - Texture composition
  - Per-pixel operations like per-pixel lighting
- Geometry Shaders
  - New optional stage for Direct3D 10
  - Per-primitive operations: consumes vertices; generates strips of triangles, strips of lines, or points
  - Maximum output count is fixed as part of stage setup



## HLSL

- C-like script language for writing programmable shaders
- Direct3D 10 exclusively uses SM 4.0
  - Vertex, Geometry, and Pixel shaders must all be authored in HLSL
  - More strict shader linkage rules
    - Can't change element order
    - Unspecified elements cannot be interspersed with used elements

face tomorrow



# Example Shader

```
cbuffer cbPerFrame
{
    float4x4 g_mWorld;
    float4x4 g_mView;
    float4x4 g_mProj;
};

cbuffer cbLights
{
    Light    g_lights[8];
};

struct Light
{
    float4 Position;
    float4 Diffuse;
    float4 Specular;
    float4 Ambient;
    float4 Atten;
};

struct ColorsOutput
{
    float4 Diffuse;
    float4 Specular;
};
```

SIGGRAPH2007

# Example Shader

```
struct VSSceneIn
{
    float3 pos : POSITION;      //position of the particle
    float3 norm : NORMAL;       //velocity of the particle
    float2 tex : TEXTURE0;      //tex coords
};

struct VSSceneOut
{
    float4 pos : SV_Position;   //position
    float2 tex : TEXTURE0;      //texture coordinate
    float4 colorD : COLOR0;     //color for gouraud shading
    float4 colorS : COLOR1;     //color for specular
    float fogDist : FOGDISTANCE; //distance used for fog calc
};
```

SIGGRAPH2007

# Example Shader

```
ColorsOutput CalcLighting( float3 worldNormal, float3 worldPos, float3 cameraPos )
{
    ColorsOutput output = (ColorsOutput)0.0;
    for(int i=0; i<8; i++) {
        float3 toLight = g_lights[i].Position.xyz - worldPos;
        float lightDist = length( toLight );
        float fatten = 1.0/dot( g_lights[i].Atten, float4(1,lightDist,lightDist*lightDist,0) );
        float3 lightDir = normalize( toLight );
        float3 halfAngle = normalize( normalize(-cameraPos) + lightDir );
        output.Diffuse += max(0,dot( lightDir, worldNormal )
            * g_lights[i].Diffuse * fatten) + g_lights[i].Ambient;
        output.Specular += max(0,pow( dot( halfAngle, worldNormal ), 64 )
            * g_lights[i].Specular * fAtten );
    }
    return output;
}
```

SIGGRAPH2007

## Example Shader (Vertex Shader)

```
VSSceneOut VSScenemain(VSSceneIn input)
{
    VSSceneOut output = (VSSceneOut)0.0;

    float4 worldPos = mul( float4( input.pos, 1 ),
                           g_mWorld );
    float4 cameraPos = mul( worldPos, g_mView );
    output.pos = mul( cameraPos, g_mProj );

    //save the fog distance for later
    output.fogDist = cameraPos.z;
    ...
}
```

SIGGRAPH2007

## Example Shader (Vertex Shader)

```
...
//do gouraud lighting
float3 worldNormal = normalize( mul( input.norm,
    (float3x3)g_mWorld ) );
ColorsOutput cOut = CalcLighting( worldNormal, worldPos,
    cameraPos );

output.colorD = cOut.Diffuse;
output.colorS = cOut.Specular;
output.tex = input.tex;

return output;
}
```

SIGGRAPH2007

# Example Shader

```
struct PSSceneIn
{
    float4 pos : SV_Position;
    float2 tex : TEXTURE0;
    float4 colorD : COLOR0;
    float4 colorS : COLOR1;
    float fogDist : FOGDISTANCE;
};

#define E 2.71828

Texture2D g_txDiffuse;

SamplerState g_samLinear
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = Clamp;
    AddressV = Clamp;
};

float4 g_fogColor;
```

SIGGRAPH2007

## Example Shader (Pixel Shader)

```
float4 PSScenemain(PSSceneIn input) : SV_Target
{
    //calculate the fog factor (FOGMODE_EXP)
    float fogCoeff = 1.0 / pow( E, input.fogDist *g_fogDensity );
    float fog = clamp( fogCoeff, 0, 1 );

    //calc the color based off of the normal, textures, etc
    float4 normalColor = g_txDiffuse.Sample( g_samLinear,
        input.tex ) * input.colorD + input.colorS;

    return fog * normalColor + (1.0 - fog)*g_fogColor;
}
```

SIGGRAPH2007

## Fixed Func EMU Sample

- Many Direct3D 9 applications use a mix of fixed-function and programmable shaders
- To ease porting, the DirectX SDK includes sample shaders to emulate the legacy pipeline

face tomorrow



SIGGRAPH 2007

On modern cards, the Direct3D 9 driver is emulating fixed-function with shaders, so shaders-only is the most efficient way to code. The driver-emulated shaders are typically more complex than what you really need 99% of the time

## D3DX10

- New streamlined version of D3DX9 utility library
  - Mesh, Mesh skinning, Sprite, Font
- D3DX10Compile\* functions for HLSL
- Texture loader helper functions
- New asynchronous resource loaders
- D3DX math routines

face tomorrow  SIGGRAPH 2007



# The Direct3D 10 Pipeline

Chas. Boyd

Microsoft Corp

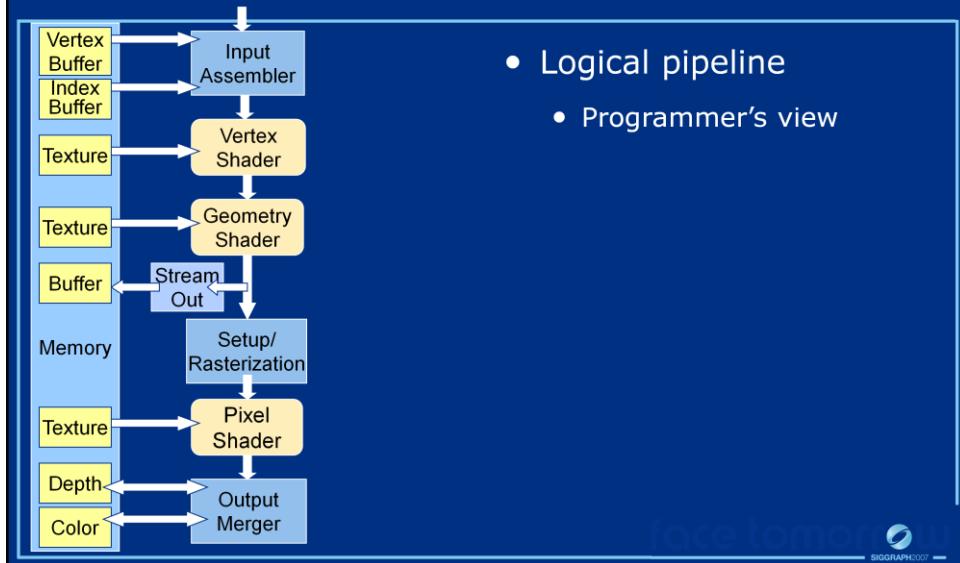
*The Direct3D 10 Pipeline* (Chas Boyd): A tour through the Direct3D 10 rendering pipeline with a detailed review of the APIs related to each stage. Comparisons with existing graphics APIs are called out to help bring the audience up to speed with the new design. [60 minutes]

## The Graphics Pipeline

Consists of a series of *programmable stages* intermixed with a series of *fixed-function stages* to produce a rendered output

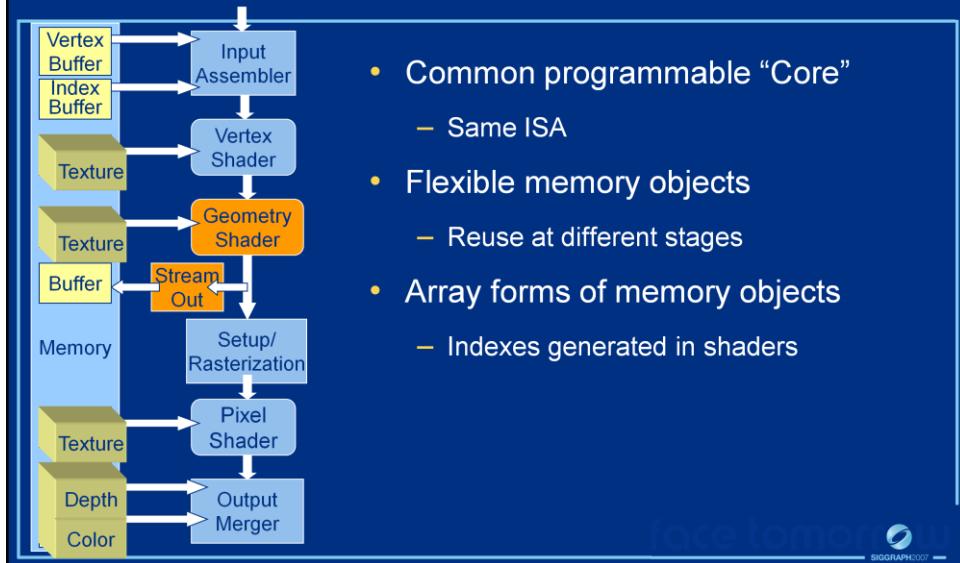
Once you understand how the pipeline works, imagine that there are many iterations of each stage running in parallel and you understand why we've segmented the way we have. In order to allow the graphics to be at the best performance available

# Direct3D 10 Pipeline



Now I'll quickly walk through the pipeline stages

# System Architecture



Now a couple of notes about the overall pipeline,

[click]

First all of the programmable stages use the same base virtual machine with the same capabilities, instruction set, and resource limits

Next, the memory mode is much more flexible. [click] Memory objects include vertex buffers, index buffers, textures, etc and they can be written to in one rendering pass and then attached to other stages of the pipeline for another rendering pass. This is a generalization of render-to-texture, render-to-vertex buffer, etc.

Finally[click], we also support an array form of the memory objects, for example a homogeneous array of 2D texture maps. When these arrays are used on the pipeline, the array indexes can be generated by the programmable stages, for example using the IDs generated by the input assembler.

## API/Runtime

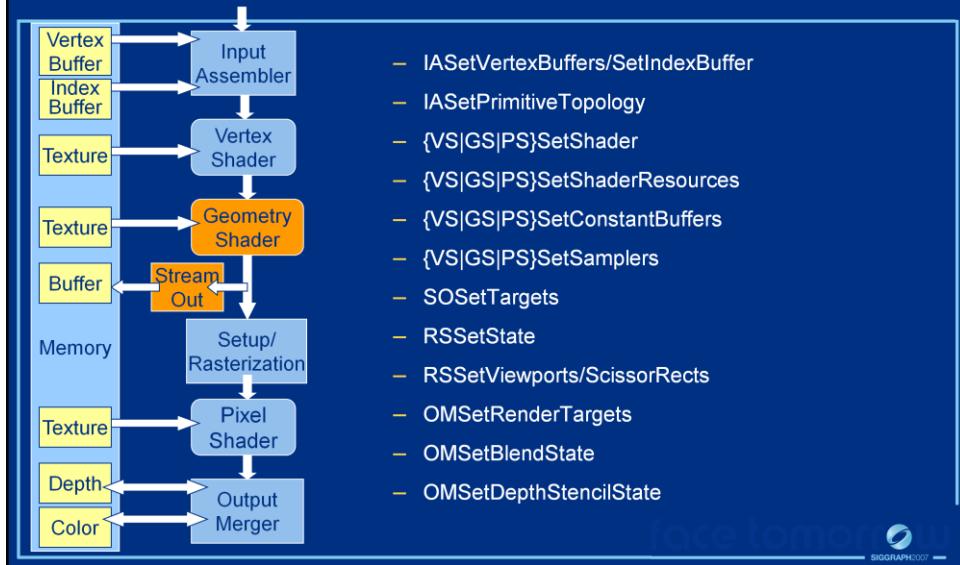
- Plumbing for
  - Creating/managing objects
  - Binding state to pipeline stages
- Restructure for efficiency & flexibility
  - Aggregate bits of state into large objects
    - More “real” work done per API call
    - Group related state together (blend, raster, stencil, depth)
    - Guide hardware implementation



In a similar vein we also restructured the API to improve efficiency. One change we made was to group related state into larger objects so that more real work is done in each API call. State that is related and likely to change together is grouped together. This grouping is also intended to guide hardware implementations to optimize these types of state updates.

This produces a slightly more object-based API model.

# Configuring the Pipeline



The end result is that configuring the entire pipeline uses 18 API calls compared to more than 100 in Direct3D 9.

We've also pushed as much validation into the creation of resources, instead of at runtime, or draw time, to minimize the software overhead

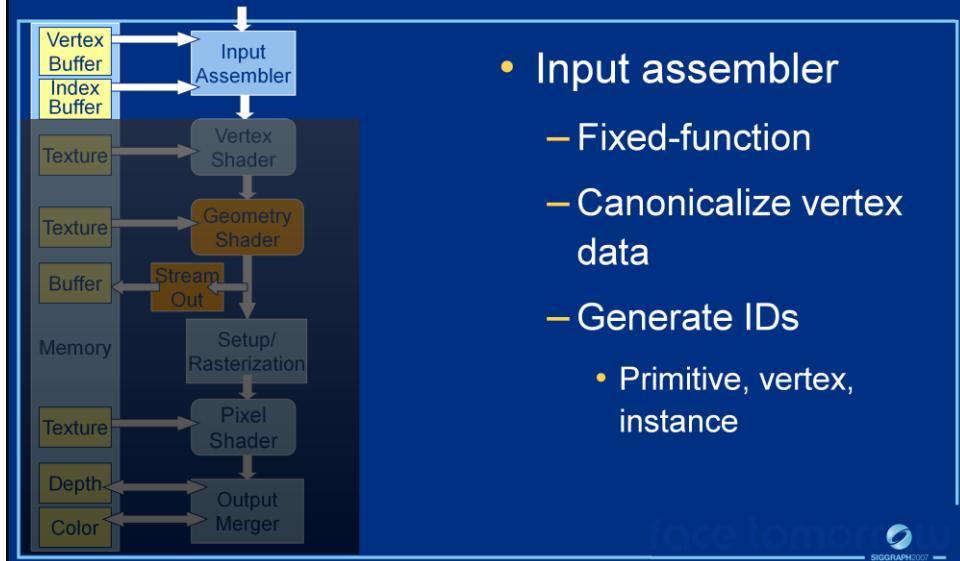
When configuring the pipeline.

# State Objects

- Reduce state-change overhead by grouping state into immutable objects
- Input Layout
  - Format, Offset, InstanceDataStepRate, ...
- Rasterizer
  - Cull Mode, Multisample Enable, Fill Mode, ...
- DepthStencil
  - Depth Enable, Depth Func, Stencil Masks, ...
- Blend
  - SrcBlend, DestBlend, BlendOp, ...
- Sampler (No longer bound to a specific texture)
  - Filter Mode, MinLOD, MaxLOD, ...

SIGGRAPH 2007

# Input Assembler



At the top, the fixed-function input assembler reads vertices and converts them to a canonical floating point representation

It also generates a unique id or index for each vertex, primitive, and object instance that can be used in other parts of the pipeline

## Input Assembler State

```
D3D10_INPUT_ELEMENT_DESC layout[] =
{
    { L"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
        D3D10_INPUT_PER_VERTEX_DATA, 0 },
    { L"TEXTURE0", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 12,
        D3D10_INPUT_PER_VERTEX_DATA, 0 },
    { L"NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 1, 20,
        D3D10_INPUT_PER_VERTEX_DATA, 0 },
};

ID3D10Device::CreateInputLayout( layout ... , &pLayout );
ID3D10Device::IASetInputLayout( pLayout );
```

SIGGRAPH2007

Input Element Descriptor describes the mapping from the input buffers to the vertex format that the vertex shader stage will consume as input.

# Input Assembler Resources

```
D3D10_BUFFER_DESC bufferDesc;
bufferDesc.Usage = D3D10_USAGE_DEFAULT;
bufferDesc.ByteWidth = sizeof( <my Vertex> ) * 3;
bufferDesc.BindFlags = D3D10_BIND_VERTEX_BUFFER;
bufferDesc.CPUAccessFlags = 0;
bufferDesc.MiscFlags = 0;

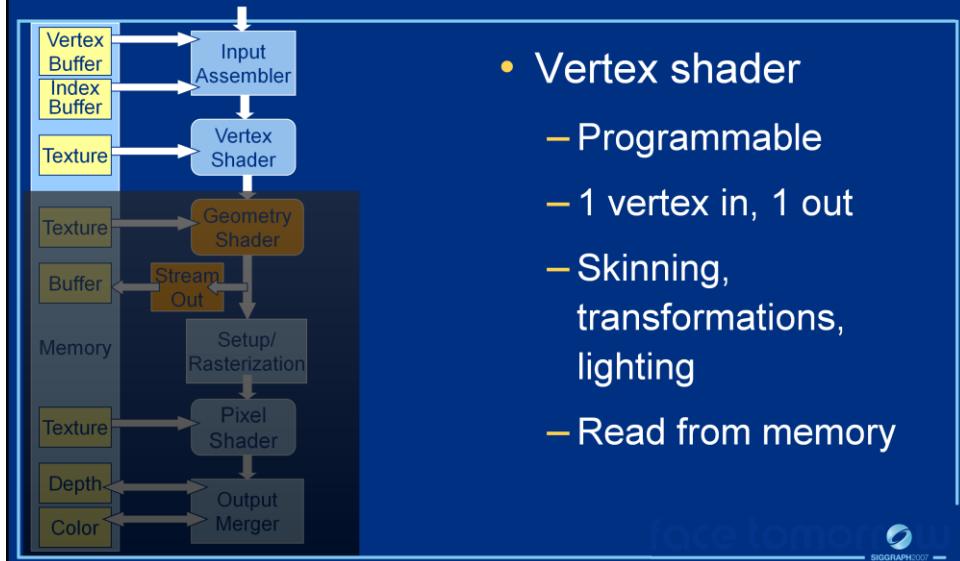
D3D10_SUBRESOURCE_DATA InitData;
InitData.pSysMem = <ptr to my vertices>;
InitData.SysMemPitch = 0;
InitData.SysMemSlicePitch = 0;

CreateBuffer( &bd, &InitData, &pVertexBuffer );
IASetVertexBuffers( ... , pVertexBuffer, ... );
```

SIGGRAPH2007

The input assembler also needs info on what resources will be read into the pipeline. A similar process is used to create indexed buffers for indexed calls. Multiple vertex buffers can be used to assemble the final vertex format which will be read by the vertex shader.

# Vertex Shader



The vertex shader is a programmable stage that reads a single vertex and produces a transformed vertex

It can be used for modeling and projection transforms, animation, diffuse lighting, etc and can read textures or other data from memory

## Vertex Shader State

```
// The shader
IPD3D10Blob *pBlob;
ID3D10VertexShader *pVertexShader

D3D10CompileShader( ... , "myShader.vsh" , ... , &pBlob );
CreateVertexShader( (DWORD*)pBlob->GetBufferPointer() ,
                    pBlob->GetBufferSize() , &pVertexShader );
VSSetShader( pVertexShader );

// Samplers
D3D10_SAMPLER_DESC SamplerDesc;
VSSetSamplers( ... , pSamplers );
```

SIGGRAPH2007

# Vertex Shader Resources

```
VSSetConstantBuffers( ... , pConstantBuffer, ... ) ;

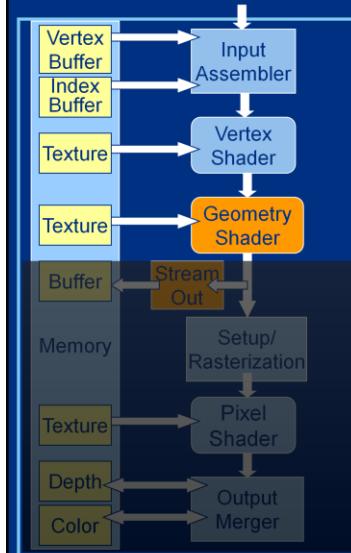
D3D10_TEXTURE2D_DESC desc;
desc.Width = 256;           desc.Height = 256;
desc.MipLevels = 1;         desc.ArraySize = 1;
desc.Format = DXGI_FORMAT_R32G32B32A32_FLOAT;
desc.SampleDesc.Count = 1;
desc.Usage = D3D10_USAGE_DEFAULT;
desc.BindFlags = < >;
ID3D10Texture2D pTexture2D;
CreateTexture2D( &desc, NULL, &pTexture2D );

VSSetShaderResources( dwStart, nResources, pResources );
```

SIGGRAPH2007

There is actually an intervening step not shown here. Resources need to be operated on in terms of ResourceViews which I talk about later on.

# Geometry Shader



- Geometry Shader
  - New, programmable
  - Per-primitive processing
  - 1 prim in,  $k$  prims out
  - Read from memory

The geometry shader is a new programmable stage that performs per-primitive processing

It reads the vertices of a point, line, or triangle primitive and produces one or more output primitives

It also has the ability to read from textures and buffers in memory

SIGGRAPH2007

## Geometry Shader State

```
// The shader
IPD3D10Blob *pBlob;
ID3D10GeometryShader *pGeometryShader;

D3D10CompileShader( ... , "myShader.gsh" , ... , &pBlob );
CreateGeometryShader( (DWORD*)pBlob->GetBufferPointer() ,
                     pBlob->GetBufferSize() , &pGeometryShader );
GSSetShader( pGeometryShader );

// Samplers
D3D10_SAMPLER_DESC SamplerDesc;
GSSetSamplers( ... , pSamplers );
```

SIGGRAPH2007

## Geometry Shader Resources

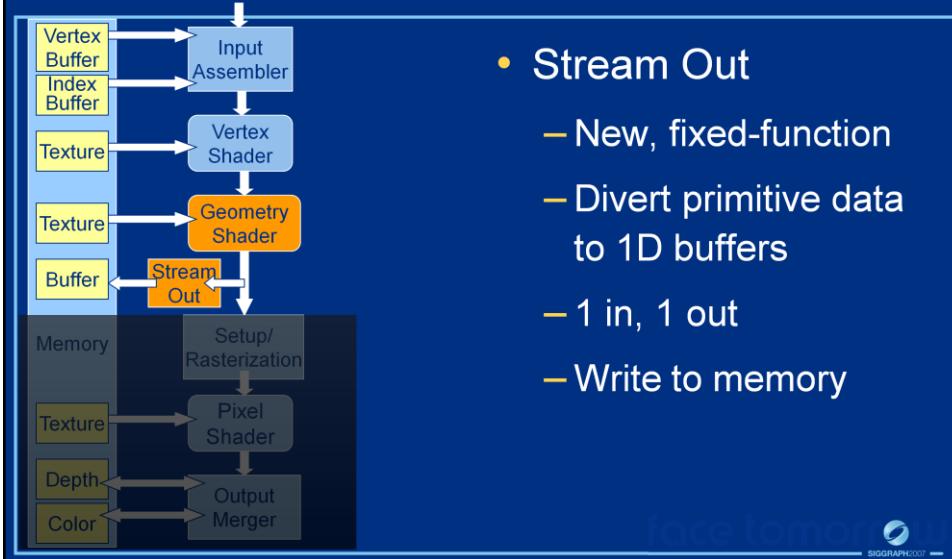
```
GSSetConstantBuffers( ... , pConstantBuffer, ... ) ;

D3D10_TEXTURE2D_DESC desc ;
desc.Width = 256;           desc.Height = 256;
desc.MipLevels = 1;         desc.ArraySize = 1;
desc.Format = DXGI_FORMAT_R32G32B32A32_FLOAT;
desc.SampleDesc.Count = 1;
desc.Usage = D3D10_USAGE_DEFAULT;
desc.BindFlags = < >;
ID3D10Texture2D pTexture2D;
CreateTexture2D( &desc, NULL, &pTexture2D );

GSSetShaderResources( dwStart, nResources, pResources );
```

SIGGRAPH2007

## Stream Out



The stream output stage is a new fixed-function stage that can write vertex data to memory

## Stream Out State

```
D3D10_STREAM_OUTPUT_DECLARATION_ENTRY pDecl[] =
{
    // semantic name, index, start, count, output slot
    { L"SV_POSITION", 0, 0, 4, 0 }, // all components
    { L"TEXCOORD0", 0, 0, 3, 0 }, // first 3 of normal
    { L"TEXCOORD1", 0, 0, 2, 0 }, // first 2 texture coords
};

D3D10Device->CreateGeometryShaderWithStreamOut(
    pShaderBytecode, pDecl, 3, sizeof(pDecl), &pGS );
```

SIGGRAPH2007

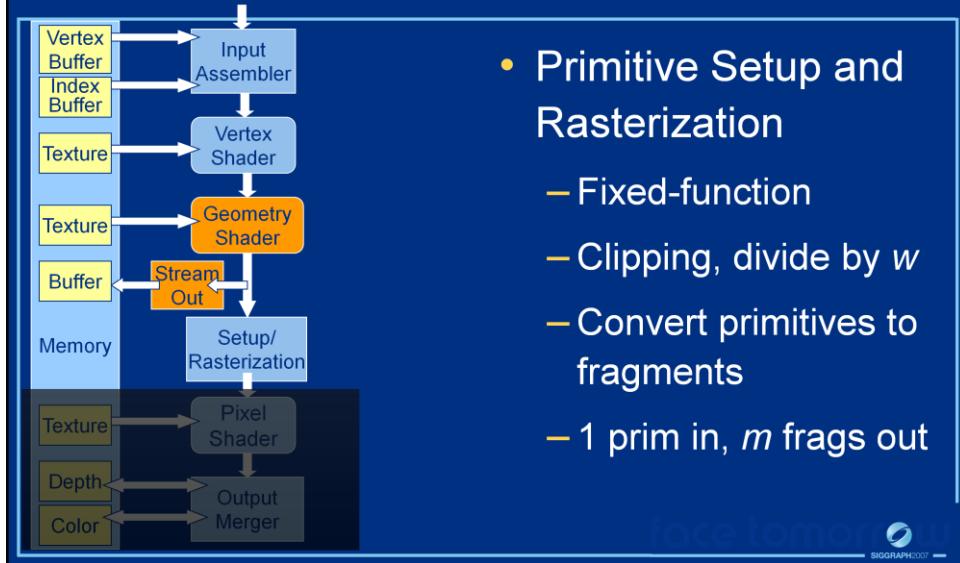
## Stream Out Resources

```
ID3D10Buffer *m_pBuffer;
int m_nBufferSize = 1000000;
D3D10_BUFFER_DESC bufferDesc =
{
    m_nBufferSize,
    D3D10_USAGE_DEFAULT,
    D3D10_BIND_STREAM_OUTPUT,
    0,
    0
};
CreateBuffer( &bufferDesc, NULL, &m_pBuffer );

UINT offset[1] = 0;
SOSetTargets( 1, &m_pBuffer, offset );
```

SIGGRAPH2007

## Setup/Rasterizer



The setup and rasterization stage performs a number of functions including clipping, homogeneous divide, triangle setup (plane equation or gradient computations), and scan conversion to convert a single primitive into a series of pixel fragments

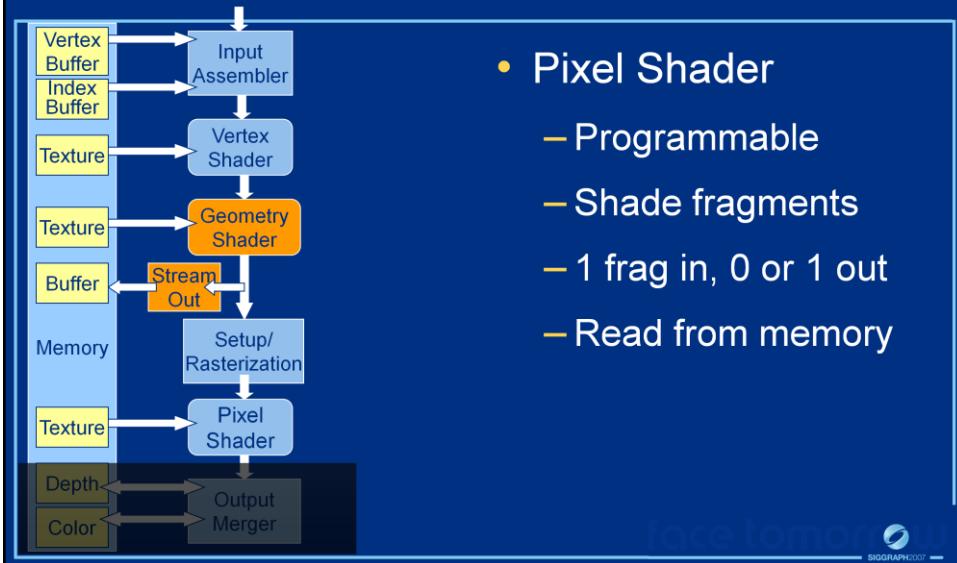
## Setup/Rasterizer State

```
D3D10_RASTERIZER_DESC rastState;  
rastState.FillMode = D3D10_FILL_SOLID;  
rastState.CullMode = D3D10_CULL_FRONT;  
rastState.FrontCounterClockwise = true;  
rastState.DepthBias = false;  
rastState.DepthBiasClamp = 0;  
rastState.SlopeScaledDepthBias = 0;  
rastState.DepthClipEnable = true;  
rastState.ScissorEnable = true;  
rastState.MultisampleEnable = false;  
rastState.AntialiasedLineEnable = false;  
  
ID3D10RasterizerState *pRasterState;  
CreateRasterizerState( &rastState, &pRasterState );  
RSSetState(g_pRasterState);
```

SIGGRAPH2007

The Setup/Rasterization stage needs to have a semantic provided to identify position so that the clipper can operate correctly.

# Pixel Shader



The pixel shader is a programmable stage that operates on a single fragment, performing shading operations. The pixel shader can read from memory to perform texturing and other operations

## Pixel Shader State

```
// The shader  
  
IPD3D10Blob *pBlob;  
  
ID3D10PixelShader *pPixelShader;  
  
D3D10CompileShader( ... , "myShader.psh" , ... , &pBlob );  
  
CreatePixelShader( (DWORD*)pBlob->GetBufferPointer() ,  
                  pBlob->GetBufferSize() , &pPixelShader );  
  
PSSetShader( pPixelShader );
```

SIGGRAPH2007

The primary element of the pixel shader state is the pixel shader itself,  
Here we compile a pixel shader, create it, and then send it to the device (make it current).

## Pixel Shader State -samplers

```
D3D10_SAMPLER_DESC SamplerDesc;  
SamplerDesc.Filter = D3D10_FILTER_MIN_MAG_MIP_LINEAR;  
SamplerDesc.AddressU = D3D10_TEXTURE_ADDRESS_CLAMP;  
SamplerDesc.AddressV = D3D10_TEXTURE_ADDRESS_CLAMP;  
SamplerDesc.AddressW = D3D10_TEXTURE_ADDRESS_CLAMP;  
SamplerDesc.MipLODBias = 0;  
SamplerDesc.MaxAnisotropy = 1;  
SamplerDesc.ComparisonFunc = D3D10_COMPARISON_NEVER;  
SamplerDesc.BorderColor[0-3] = 1.0f;  
SamplerDesc.MinLOD = -FLT_MAX;  
SamplerDesc.MaxLOD = FLT_MAX;  
  
D3D10_SAMPLER_DESC pSamplers[0] = SamplerDesc;  
  
PSSetSamplers( ..., pSamplers );
```

SIGGRAPH2007

# Pixel Shader Resources

```
PSSetConstantBuffers( ... , pConstantBuffer, ... ) ;

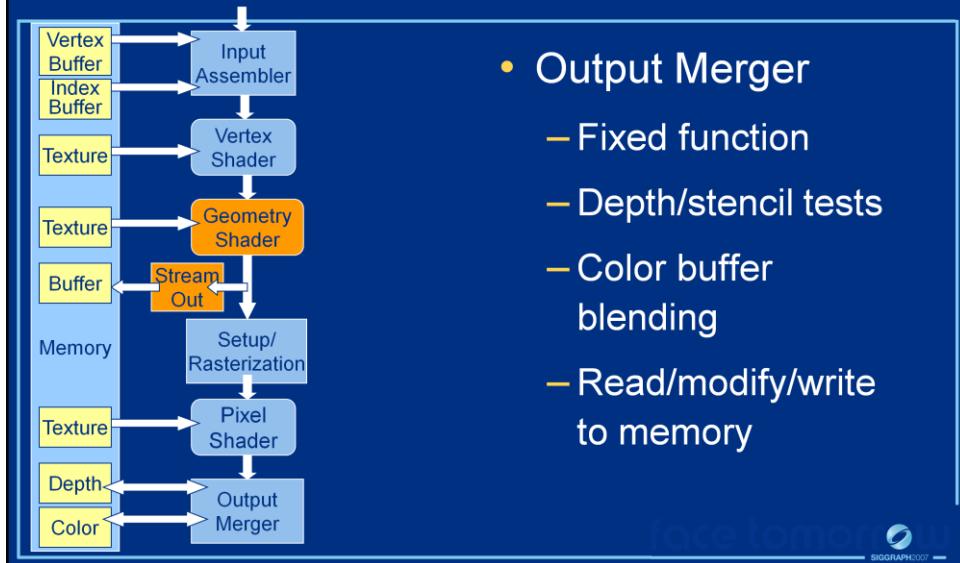
D3D10_TEXTURE2D_DESC desc;
desc.Width = 256;           desc.Height = 256;
desc.MipLevels = 1;         desc.ArraySize = 1;
desc.Format = DXGI_FORMAT_R32G32B32A32_FLOAT;
desc.SampleDesc.Count = 1;
desc.Usage = D3D10_USAGE_DEFAULT;
desc.BindFlags = < >;
ID3D10Texture2D pTexture2D;
CreateTexture2D( &desc, NULL, &pTexture2D );

PSSetShaderResources( dwStart, nResources, pResources );
```

SIGGRAPH2007

There is actually an intervening step not shown here. Resources need to be operated on in terms of ResourceViews which I talk about later on.

# Output Merger



At the bottom of the pipeline is a fixed function stage that performs depth and stencil testing as well as color blending operations. This is sometimes referred to as the 'back end', or the ROP (raster op) unit.

Read-modify-write memory operations are necessary to implement this functionality and output merger stage is the only stage with this capability.

## Output Merger State -blending

```
D3D10_BLEND_DESC BlendState;  
ZeroMemory(&BlendState, sizeof(D3D10_BLEND_DESC));  
BlendState.BlendEnable[0] = FALSE;  
BlendState.RenderTargetWriteMask[0] =  
    D3D10_COLOR_WRITE_ENABLE_ALL;  
  
ID3D10BlendState* pBlendState = NULL;  
CreateBlendState(&BlendState, &pBlendState);  
  
float blendFactor = 0;  
UINT sampleMask = 0xffffffff;  
OMSetBlendState(pBlendState, blendFactor, sampleMask);
```

SIGGRAPH2007

The Output Merger has state grouped into 2 objects.

This state object handles the blending operations.

First we create the blend state object from the blend descriptor struct,

Then, in the main draw loop, we set it to the device,

along with a couple other parameters that are likely to be updated more frequently than the rest of the blend state.

## Output Merger State -depth

```
D3D10_DEPTH_STENCIL_DESC dsDesc;
// Depth test parameters
dsDesc.DepthEnable = true;
dsDesc.DepthWriteMask = D3D10_DEPTH_WRITE_MASK_ALL;
dsDesc.DepthFunc = D3D10_COMPARISON_LESS;
// Stencil test parameters
dsDesc.StencilEnable = true;
...
// Create depth stencil state
ID3D10DepthStencilState *pDSState;
CreateDepthStencilState(dsDesc, &pDSState);
OMSetDepthStencilState(pDSState, 1);
```

SIGGRAPH2007

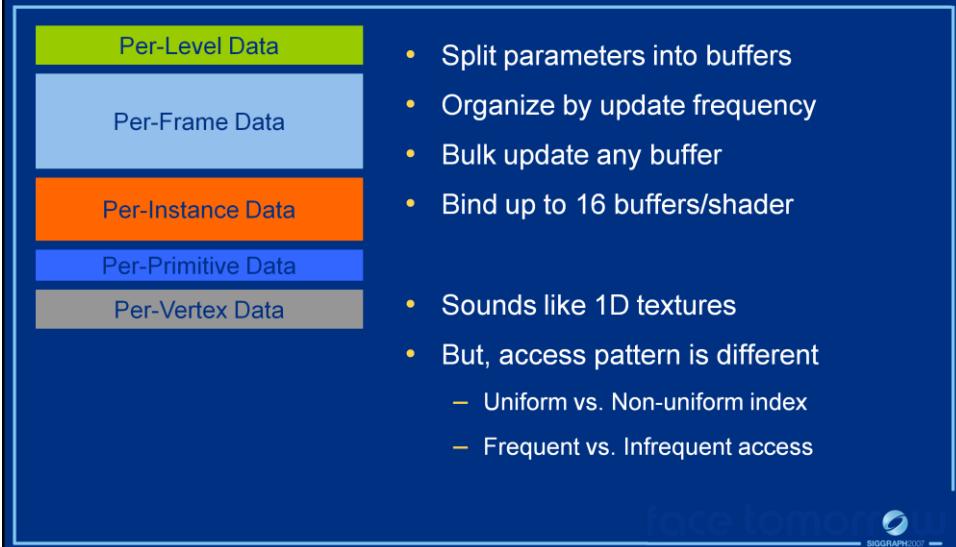
The output Merger's 2<sup>nd</sup> state object is the Depth/Stencil state.

We create the object from the struct, then at runtime, we set it on the device to make it current.

## Resources & Buffers

- Textures aren't just textures anymore
  - Normal maps, lookup tables, render targets, etc.
- Resources allow for generalized usage of data
- Same goes for buffers for vertices and indices

# Constant Buffers



In Direct3D 10 we partition this parameter state into separate buffers organized by update frequency.

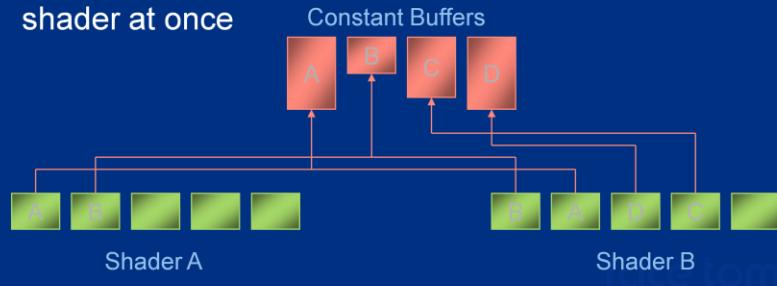
Besides being larger, these buffers can be independently updated and bound to the pipeline

At first we thought we could use 1D textures as these buffers and simplify the model. But we noticed that the access patterns are different from textures in that constants are usually indexed uniformly – all shaders look at the same projection matrix, and that they are accessed at much higher instruction frequency without filtering

This led us to make constant buffers be a separate type of resource.

# Constant Buffers

- Constants now managed like vertex/texture data
  - Updated efficiently via lock/discard or UpdateResourceUP
  - Set like any other resource
- Up to 4096 4-channel × 32-bit elements per CB
- Create as many CBs as you want; 16 can be bound to a shader at once



As shader length and complexity increases, it becomes important to be able to assign many constants to a shader at once.

# Constant Buffers

- Example HLSL Syntax

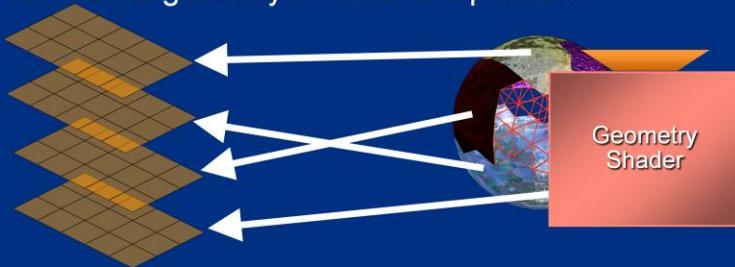
```
cbuffer myObject
{
    float4x4 matWorld;
    float3 vObjectPosition;
    int     arrayIndex;
}

cbuffer myScene
{
    float3 vSunPosition;
    float4x4 matView;
}
```

- Variables still exist in the global namespace
  - arrayIndex = 4;
  - myObject.arrayIndex = 4;

## New Resource Type: Texture Arrays

- Dynamically indexable in the Shader
- Whole array can be set as a render target
  - GS Can Emit a System-Interpreted Value:  RenderTargetArrayIndex for the primitive



SIGGRAPH 2007

## Resource Views

R32\_TYPELESS

01001011101000101100011010010100

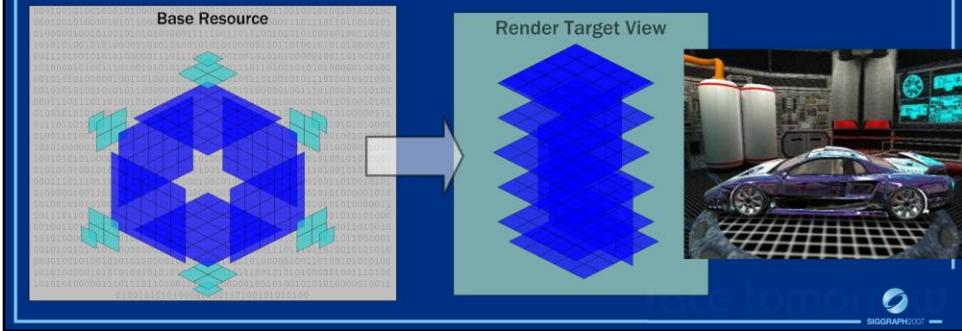
- Resources in DX10 are generally typeless
- Resource must be interpreted as a specific type by obtaining a **view** of the resource
- Allows you to reinterpret data in a different format
- Forces type validation earlier in setup
  - Don't have to re-validate on every draw

Choice in DX9 was between structured data, and void\* which could not be validated.  
Views provide a polymorphism mechanism that can still be validated.  
Currently dimensions and element size must be consistent between all views.

# Resource Views

## Resource Views Example: Cubemap

- Views enable interpretation of resources at different bind locations



The same memory can be interpreted as an array of textures (such as used in a render target), and later as a cube-map (used by a texture sampling shader).

## The Shader Core

- New Unified Shader Core
  - All shader stages use the same cores
  - Have the same functionality
- Comparison-Sample instruction
  - Percentage-Closer shadow Filtering
- Immediate offset (up to +/-8) on Texture/Buffer load
  - Custom filter kernels
- Resource Info
  - Returns height, width, # of miplevels, arraysize for the resource view
- More of everything
  - Inter-stage registers, samplers, textures
  - Unlimited instruction count

face tomorrow



A major effort was undertaken to improve consistency between different shader types. This enables easier programming, as well as a unified implementation. The resource information enables reflection so that the app can identify key information about a shader.

# The Virtual Machine

	Direct3D 9	Direct3D 10
Instructions	64K/512	unlimited
Textures	16	128
Temporary registers	16	4096 float4s
Constants	256	4kx16 float4s
Interstage registers	16	32
2D texture	4Kx4K	8Kx8K
Render targets	4	8

- New Features

- Integer instruction set
- Load instruction (no store!)
- IEEE-754 format & ~accuracy
- Separate samplers & textures
- Writable private memory

The table compares some of the resource limits between the previous generation and Direct3D 10

We'll also list a few of the features in the new virtual machine

There is a full integer instruction set including bitwise operations

There is a load or unfiltered read from memory instruction using integer addressing

And finally the floating-point format is identical to IEEE-754 used on CPUs and the accuracy of arithmetic operations is vanishingly close. We couldn't close the gap because the hardware cost in this timeframe would have been prohibitive.

Samplers – which control address modes and filtering properties – have been decoupled from actual textures, since they vary at a lower frequency.

There has been a significant increase in the temporary data that can be used in a shader invocation.

## HLSL 10

- Direct3D10 shader authoring in HLSL
  - Minimizes invalid state introduced by ASM model
  - Minimizes redundant intermediate representations
  - HLSL optimizations guaranteed to the driver
  - Enables fast shader linkage validation with signatures
  - Maximum asset portability
- HLSL 10 shader disassembly for debugging:

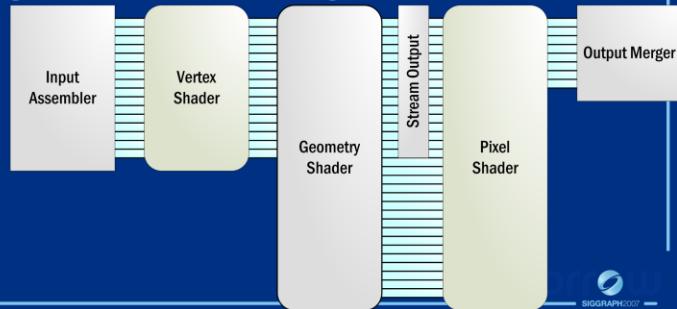
```
D3D10DisassembleShader(CONST void *pShader, ...,
    LPCSTR pComments, ID3D10Blob **ppDisassembly );
```
- Author-time compilation still supported

Just hand in the compiled shader to



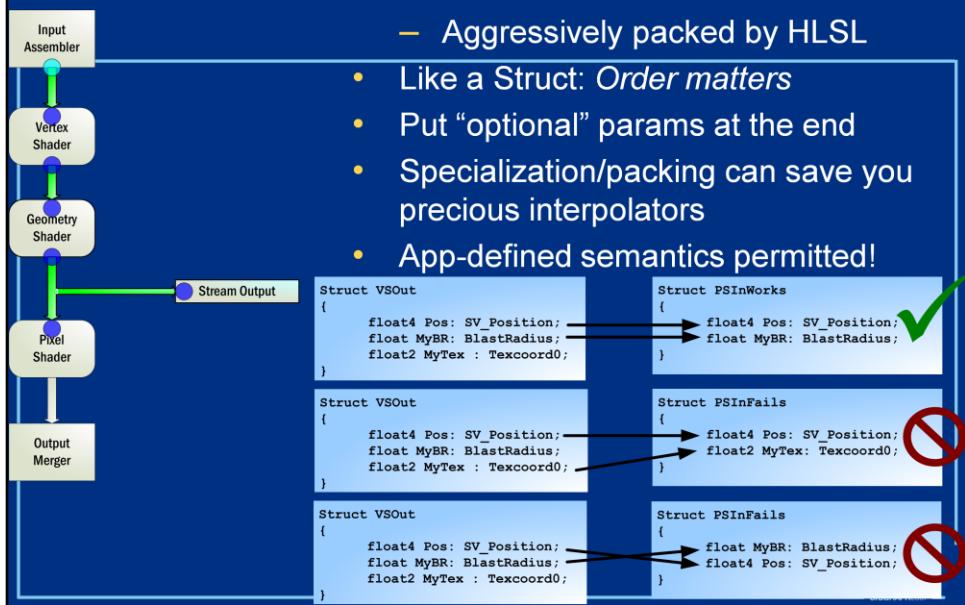
## Fast Interstage Linkage

- Direct3D10 API Design Imperative:  
No draw-time fix-up required
- Fixed generic register back connects stages
- Shader linkage enforce via ‘Signatures’



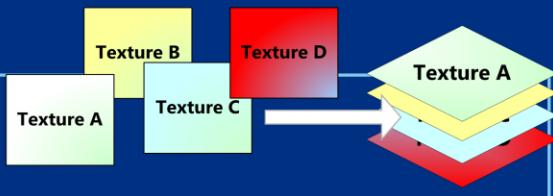
# Signatures

- Correspond to shader parameter declarations
  - Aggressively packed by HLSL
- Like a Struct: *Order matters*
- Put “optional” params at the end
- Specialization/packing can save you precious interpolators
- App-defined semantics permitted!

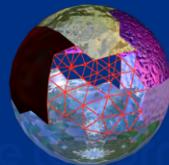


# GPU Material System

- Using the GS, delegate material properties to the Pixel Shader per-primitive
- Put all your similar-sized textures into a single array
- Use a switch statement to group material code
- Put material management onto GPU!



```
Switch(MatID)
{
    Pixel Shading Function
    Pixel Shading Function
}
```



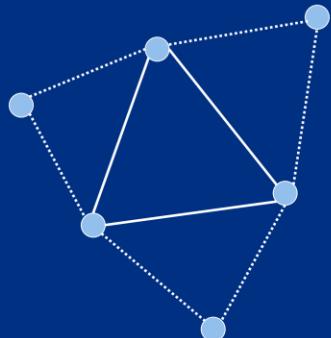
SIGGRAPH2007

## Material System Flexibility

- This may require some complex shader execution
  - Eats up registers
  - More GPU thread state
- Shader Specialization is still very powerful
- D3D10 allows **YOU** to choose what is processed on the CPU or on the GPU
- Author on Direct3D 10 →  
Specialize for performance and cross-platform



# Geometry Shader



- Entire primitive as input
  - Adjacency Optional
- Outputs zero or more primitives
  - 1024 scalars out max

face tomorrow SIGGRAPH 2007

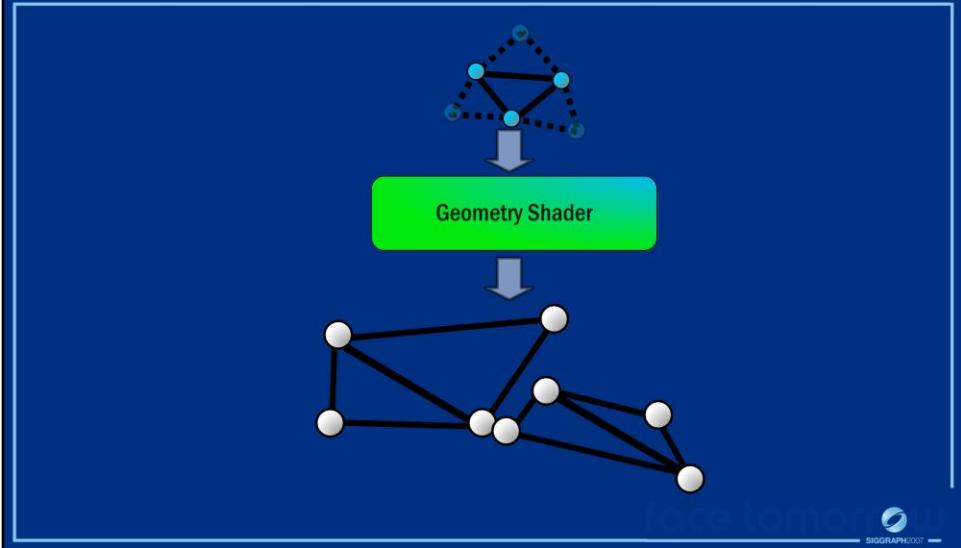
Some more details about the geometry shader.

Its input is an entire primitive (point, line or triangle) and it outputs a homogenous list of points/lines or triangles – a maximum of 1k scalars.

[click]

You can optionally have adjacent indices as inputs, which makes things like edge extrusion for shadow techniques easier.

# Geometry Shader - Primitive Setup



SIGGRAPH 2007

# Primitive Topologies

- New Primitive Topologies
  - Include Adjacency

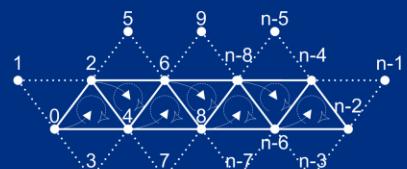
Line List w/ adjacency



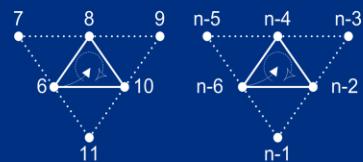
Line Strip w/ adjacency



Triangle Strip w/ adjacency



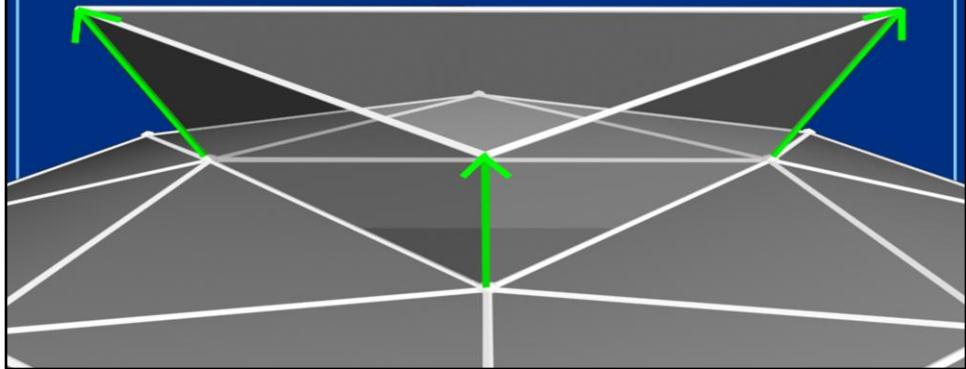
Triangle List w/ adjacency



face tomorrow SIGGRAPH2007

## Geometry Shader - Primitive Setup

- Take control of attribute evaluation and interpolation!
  - Extrude the triangle for depth



# Geometry Shader – Primitive Setup

- Take control of attribute evaluation and interpolation!
  - Extrude the triangle for depth
  - Add control points for higher-order interpolation

Example basis functions:

$$2x^2 + 2y^2 + 4xy - 3x - 3y + 1$$

$$-4x^2 - 4xy + 4x$$

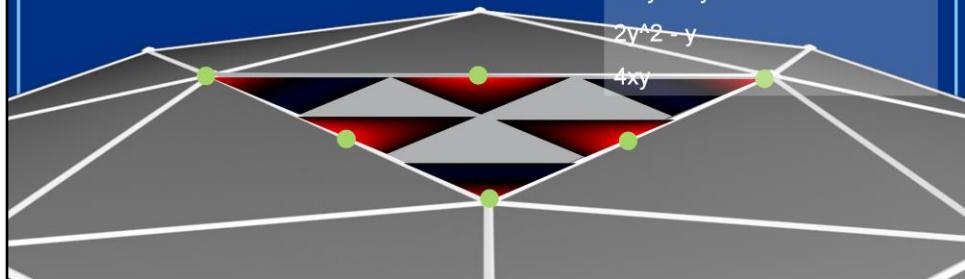
$$2x^2 - x$$

$$-4y^2$$

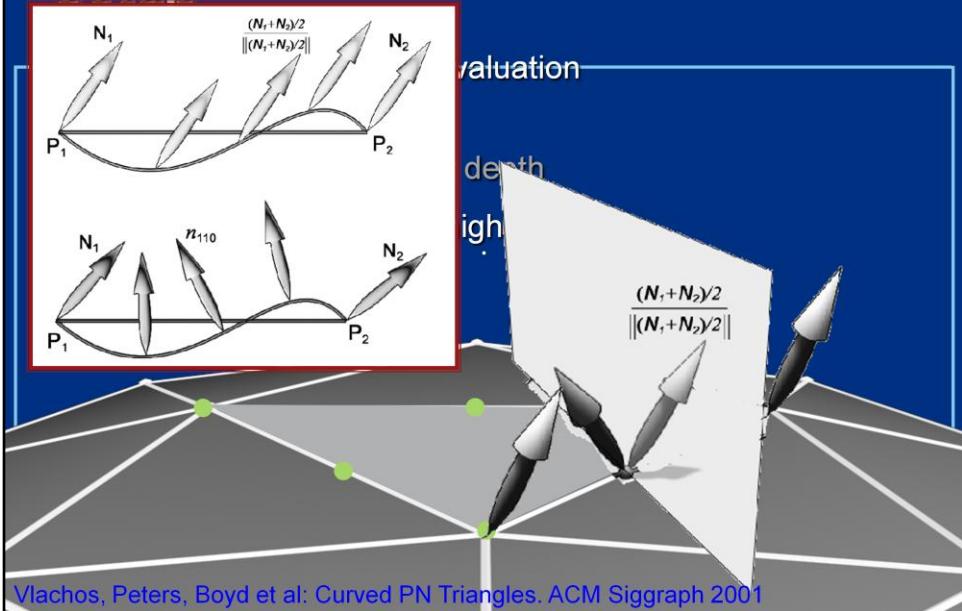
$$-4xy + 4y$$

$$2y^2 - y$$

$$4xy$$

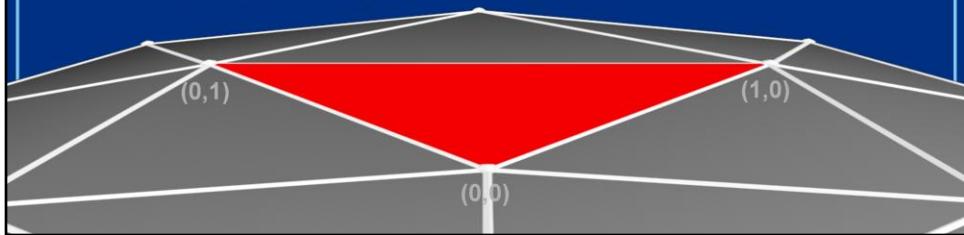


# Geometry Shader - Primitive Setup



## Geometry Shader - Primitive Setup

- Take control of attribute evaluation and interpolation!
  - Extrude the triangle for depth
  - Add control points for higher-order interpolation
  - Compute plane equations/Barycentrics



# Determinism & Parallelism

- Allow parallel processing but preserve serial order
- 
- ```
graph TD; 1[1] --> GS1[GS]; 2[2] --> GS2[GS]; ...[...] --> GSn[GS]; GS1 --> T1[1]; GS2 --> T2[2]; ...[...] --> Tn[n];
```
- Buffer GS outputs (on chip)
  - Limit output to 1K 32-bit values
  - Application can specify less
    - May allow greater parallelism

Expansion  
to 2 triangles

One final note about the geometry shader

We stated that we want to support parallel processing. The output of the geometry shader is defined to preserve the order of the input primitives.

This constraint requires that the output of parallel elements be buffered to maintain the correct output order.

To avoid adding too much hardware cost, we limit the total amount of output that can be generated from a single primitive to 1024 32-bit values.

The geometry shader isn't really intended for large scale amplification.

# Geometry Shader - Primitive Setup

## Computing plane equations/Barycentrics

(0,1)



(1,0)

(0,0)

```
vert0 = Load(..., vertID[0])  
vert1 = Load(..., vertID[1])  
vert2 = Load(..., vertID[2])
```

Pixel Shading Function

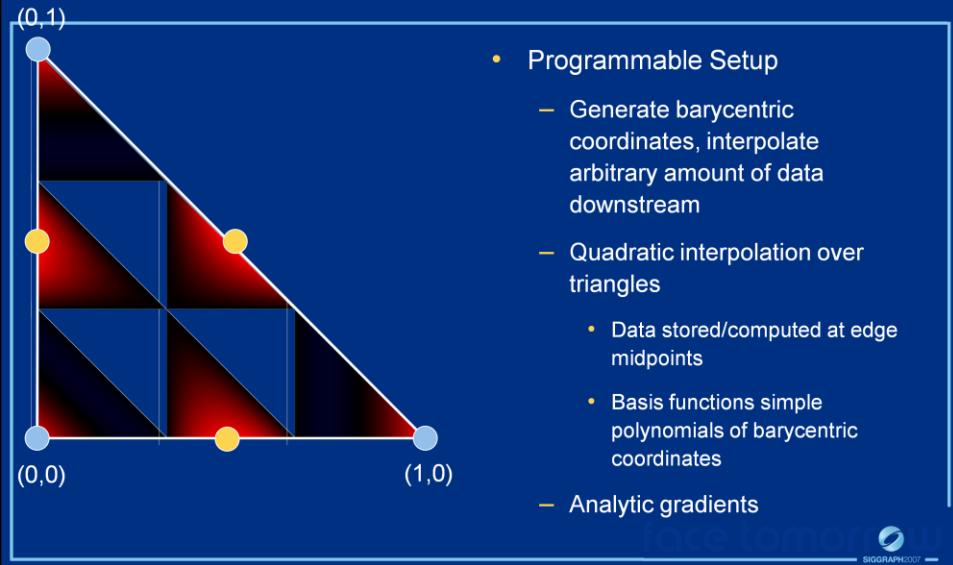
Per-Vertex Data

- Exceed max interpolators (16 out of VS, 32 out of GS)
- Anti-alias:
  - Generate high-quality derivatives anywhere
  - Interpolate outside of the pixel center/centroid
    - Don't jump out-of-gamut when texture filtering!



SIGGRAPH

## Geometry Shader



One role that the GS can be used for is programmable setup – where you are still outputting single primitives for each input primitive, you are just using This stage to set up some type of ancillary data.

For example you can generate the barycentric coordinates (which can then be used in the pixel shader) by simply assigning them at the vertices Of a triangle.

[click]

Here we see them specified for the two highlighted vertices, the third can just be computed based on these in the pixel shader.

These can be used for many things, one example would be to also send vertex indices down to the pixel shader and interpolate arbitrary Amounts of data. The only way to do this in DX9 would be to blow out your triangles individually, which would be extremely expensive if you are using fat vertices.

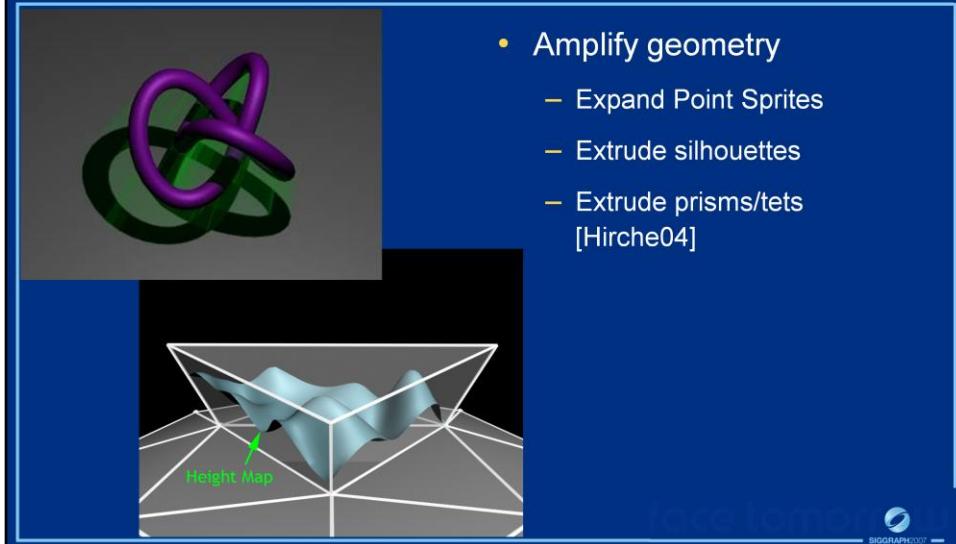
[click]

You can also implement higher order interpolation over planar triangles, for quadratic interpolation you just need to [click] store or compute values at The edge midpoints, and send them as constants. These 6 basis functions are simple polynomials of the barycentric coordinates.

This can be used for better interpolation of normals over triangles, or other smooth functions – for example incident radiance or transfer coefficients.

You can also use the GS to setup the computation of analytic derivatives in the pixel shader.

# Geometry Shader



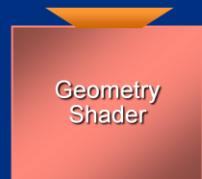
The GS can also be used to amplify geometry. Fixed function point sprites have been removed, because they are so easy to do in the GS. You just take points In and output triangles.

As I mentioned before using the adjacency information you can extrude silhouette edges in the GS for shadow volumes.

Another example along these lines is recent techniques that do displacement mapping by ray tracing heightfields – I'll briefly discuss an example of this later.

# Geometry Shader

## Amplification and De-Amplification



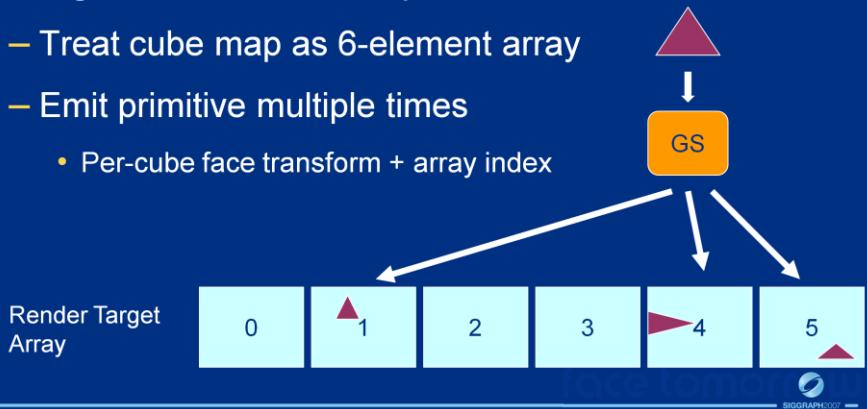
- Emits primitives of a specified output type (point, linestrip, trianglestrip)
  - *Limited* geometry amplification/de-amplification:  
Output 0-1024 values per invocation
- No more 1-in / 1-out limit!
  - Shadow Volumes
  - Fur/Fins
  - Procedural Geometry/Detailing
  - All-GPU Particle Systems
  - Point Sprites

face tomorrow



# Geometry Shader

- Generate Array Index for render target array
  - E.g., render to cube map
  - Treat cube map as 6-element array
  - Emit primitive multiple times
    - Per-cube face transform + array index



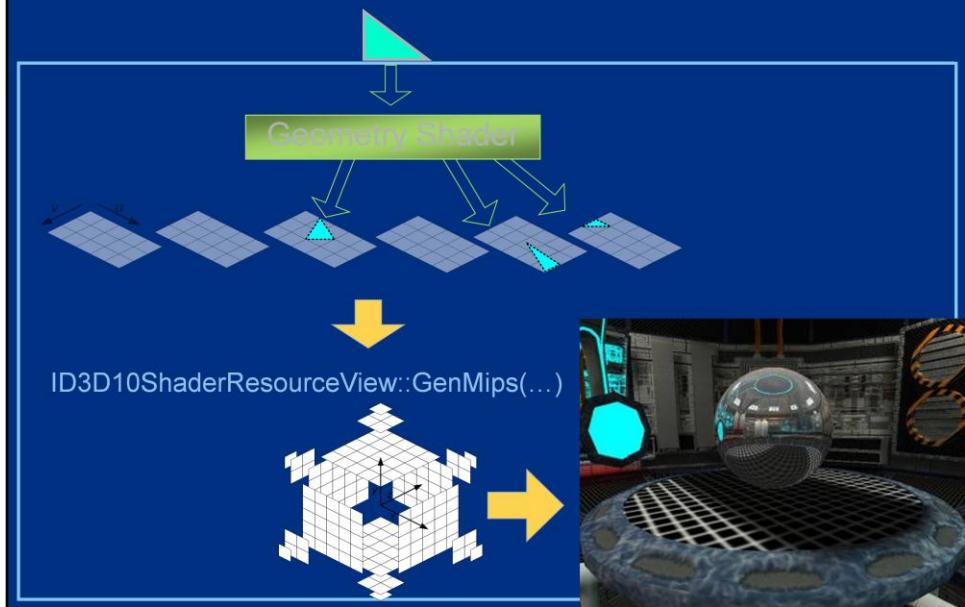
The geometry shader can also generate an index for addressing an array of color buffers at the output merger (we call them render targets)

We can combine this with the multiple output capability to render to a cube map in a single rendering pass

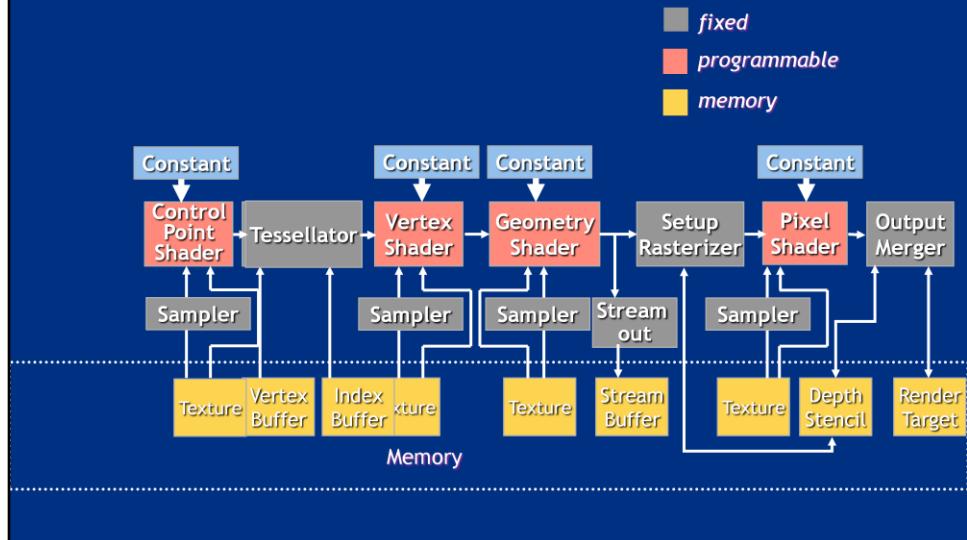
[click]

The 6-face cube map is treated as a 6-element array. Each input primitive is transformed by a per-cube-face transform and output to the appropriate cube face

# Single-Pass Render-to-Cubemap



# (Logical) Pipeline Evolution



## Summary

- Use pipeline as a mental model when constructing a renderer
- Think ‘object oriented’ when managing state
- New stages may be added in later releases
- It’s all documented in the help file





*HLSL Shader Model 4.0* (Michael Oneppo): Learn about advancements in the HLSL language to support the more general and robust programming model in Shader Model 4.0. This talk covers use of geometry shaders, integer instructions, new texture intrinsics and flow control mechanisms. Techniques for developing shader content to be shared between Direct3D9 and 10 are also discussed. [60 minutes]

## Overview

- Review of HLSL
- Advancements in HLSL for D3D10
- Utilizing HLSL in your pipeline
- Using HLSL for D3D10 and D3D9
- Best Practices
- Conclusion



# HLSL Evolution

- HLSL evolves in-sync with hardware shader models
- Massive programmability increase in D3D10 shader model 4.0 has enabled new language enhancements
- New language features are emulated down-level where possible
- Developer feedback is a key factor
- Making the language easier to use
- Making decisions the compiler has to make more transparent

face tomorrow 

## HLSL Review

- HLSL is a highly effective language for writing shaders
- Syntax is similar to 'C'
  - Preprocessor defines (#define, #ifdef, etc)
  - Basic types (float, int, uint, bool, etc)
  - Operators, variables, functions
- Has some important differences
  - Built-in variables (float4, matrix, etc)
  - Intrinsic functions (mul, normalize, etc)



The built-in variable types and intrinsic functions are the first peek you get at the major difference between HLSL and C-based languages. HLSL is designed to work with SIMD architectures whose atomic operations work on multi-dimensional types.

## HLSL Review

- Compiler (**fxc.exe** or D3DX library) generates target-specific instructions from shader.
  - Highly-optimized, validated Intermediate Representation is consumed by the driver
  - Different instruction sets for different generations of hardware



For Direct3D 10, an intermediary representation exists that is compiled to the native hardware instruction set. In previous targets, SM 3.0 and below, the compiler generates the instructions directly.

## HLSL Compiler Benefits

- Arbitrary, easily-editable, high-level shader code – hand-authored or generated from a tool – is automatically streamlined and optimized into a compiled representation that runs efficiently in hardware.
  - Vectorization
  - Loop unrolling/Branch flattening and simulation
  - Dead code removal
  - Emulating array reads/writes
  - Register allocation that handles complex swizzle requirements
  - Massive suite of math and program-flow optimizations

# HLSL Advancements for D3D10

face tomorrow



## D3D10: The Shader Core

- New Unified Shader Core
  - All shader stages use the same cores
  - Have the same functionality
- Comparison-Sample instruction
  - Percentage-Closer shadow Filtering
- Immediate offset (up to +/-8) on Texture/Buffer load
  - Custom filter kernels

face tomorrow



All capabilities in all shader stages should be identical, with the exception of the stream out capabilities of the Geometry Shader which are only available to that specific stages.

Comparison sampling is new to D3D10 and we'll talk a bit on how to use it in a few slides. The immediate offset sampling adds an extra parameter to all sample instructions that is an offset, in texels, from the sampling center. This allows for very fast kernel filtering – using the same UV the shader can loop over all texels in the kernel window and do a weighted sum of the sampled values.

## D3D10: The Shader Core

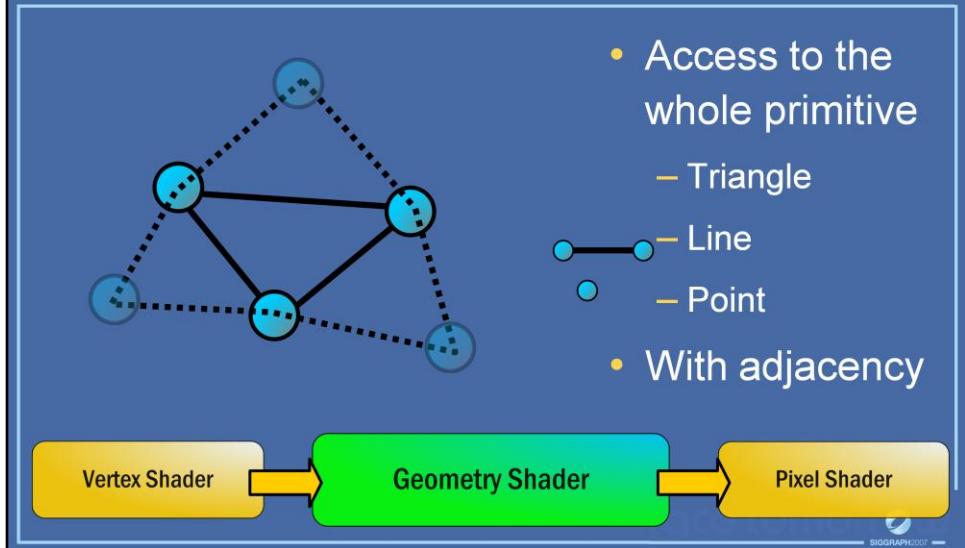
- Flexible Resource Control
  - Buffer Loads
  - Indexable textures
  - Indexable constants
  - Indexable temps
  - Indexable outputs
- Resource Info
  - Returns height, width, # of miplevels, arraysize for the resource view
- More of everything
  - Inter-stage registers, samplers, textures
  - Unlimited instruction count

face tomorrow



SIGGRAPH 2007

# The New Pipeline Direct3D10 – Geometry Shader



The GS is primarily designed for primitive setup. Most of the utility comes from being able to use per-primitive data supplied by the application, adjacency information, and vertex data to make smart decisions about materials, omit whole primitives, or add a **LIMITED NUMBER** of primitives as necessary.

## HLSL 10: Goals

- Scalability: Provide a more robust language and compiler for the more complex shaders encouraged by newer hardware
- Flexibility: Design and code the way you want to - compiler will optimize it to perform best on hardware
- Visibility: Show developers what the compiler is doing

face tomorrow



HLSL has served us well for the highly specialized domain of GPU programming. As GPUs become more and more generalized, HLSL needs to be able to handle the increasing complexity developers will want to accomplish. To this end, we've been working to increase flexibility of the language and allow the scale of development sought after to be available.

## New D3D10 Shader Features

- Flow Control:
  - Branching
    - **if/else** (available pre-DX10)
    - **switch**
  - Loops: **for/while**
  - Return and Continue instructions
- Constant Buffer Organization
- New Texture Types

Basic out-of-order execution paradigms like flow control, available to CPU architectures have either been unavailable or extremely limited since they can potentially affect performance. HLSL supports the language constructs, and depending on the performance characteristics, the compiler will utilize native hardware branching instructions (available in later Direct3D 9 shader models, and Direct3D 10) or ‘flatten’ the flow control – more on this later.

Constants in Direct3D 10 are offered through Constant Buffers, which allow the developer or the compiler to sort related constants into buffers that can be updated by the application in a similar manner to textures.

## New D3D10 Shader Features

- Arbitrary semantic names
  - System-Interpreted and System-Generated semantics start with SV\_
    - Position, VPos <-> SV\_Position
    - Color <-> SV\_Target
    - VFace <-> SV\_IsFrontFace
- Backward compatibility mode allows using the old names in 4.0 shaders
- D3D9 profiles support new semantic name 

Direct3D 10 compiler offers arbitrary semantics and annotations. There are a few values used by the various shader stages that are provided by the hardware, and to access these, there are a few system-generated semantics.

## Giving Control Back to the Programmer: Attributes

- We've added an attribute system that allows you to assist the compiler in making certain code generation decisions
  - E.g., unrolling loops and removing branches
- Syntax

```
[unroll(3)] for( ... )  
[flatten] if( ... )
```

face tomorrow  SIGGRAPH 2007

Flattening a control structure means that the compiler will emit code to compute both paths and assign the appropriate values afterward. Unrolling loops is as straightforward as just duplicating the code the number of times indicated.

This is extremely powerful in optimizing shaders during development. Afraid a loop is affecting your performance? Unroll it and see what happens.

# Flow Control Attributes

- Loop attributes
  - **[unroll(n)]** - Force loop unrolling, up to **n** iterations of the loop
  - **[loop]** - Force not to unroll
- If statement attributes
  - **[branch]** - Force branching
  - **[flatten]** – Force no branches

# Constant Buffers

- Example HLSL Syntax

```
cbuffer myObject
{
    float4x4 matWorld;
    float3 vObjectPosition;
    int     arrayIndex;
}

cbuffer myScene
{
    float3 vSunPosition;
    float4x4 matView;
}
```

- Variables still exist in the global namespace
  - ~~arrayIndex = 4;~~
  - ~~myObject.arrayIndex = 4;~~

Use cbuffer structs to define your constant buffers as you see fit. If you just use constants as globals, the compiler will organize them into constant buffers automatically, usually into a single CB.

## Texture Types

- Textures in 10 are independent of samplers
  - Can use any sampler with any texture
- Textures can be represented by object-like types
  - Texture2D, Texture3D, Texture2DArray, ...
- Texture types have methods of all sampling operations
  - Sample, SampleGrad, Load, SampleCmp
- Also have reflection data
  - GetDimensions



Texture arrays are a big addition to Direct3D with version 10.

SampleCmp takes in an additional value to compare against, and will return 1 only if the sampled value matches the provided parameter. There is also SampleCmpLevelZero, which compares the texture sample against 0.

## Texture Types: Templates

- Format of sampling is implied as float4 unless explicitly specified.
  - Exception: MSAA Textures

```
sampler MySamp;
Texture2D <float3> MyTex;
Texture2D MyTexFloat4;

...

float3 myFloat3Sample = MyTex.Sample(MySamp, TexCoords[0]);
float4 myFloat4Sample = MyTexFloat4.Sample(MySamp, TexCoords[0]);
```

## Stream Types

- Streams Types are used to output geometry from the Geometry Shader
- Three available streams for output:
  - **Pointstream**
  - **Linestream**
  - **Trianglestream**
- All act as template types:

```
TriangleStream<MyVertexStruct> myTriStream;
```

## Stream Types

```
void GS_Sample( triangle MyVertexStruct In[3],  
    inout TriangleStream<MyVertexStruct>  
    CubeMapStream )  
{  
    ...  
}  
• Streams have 2 functions for outputting  
geometry  
– Append() – takes templated type as argument  
– RestartStrip() – Resets a triangle or line strip
```

## Giving Control Back to the Programmer: Transparency

- Better information provided when the compiler unrolls loops.
  - `loop.fx(8): warning X3550: sampler array index must be a literal expression, forcing loop to unroll`
- Attributes honored or compilation fails
  - `loop.fx(7): error X3531: Can't unroll loops marked with loop attribute`

## Using HLSL 10 with D3D 10

- Direct3D10 Shader Authoring in HLSL
  - Minimizes invalid state introduced by ASM
  - Minimizes redundant intermediate representations
  - HLSL optimizations guaranteed to the driver
  - Enables fast shader linkage validation with signatures
  - Maximum asset portability
- HLSL 10 shader disassembly is available for debugging:  
D3D10DisassembleShader()
- Author-time compilation (using FXC rather than compiling shaders on the fly) still supported *and recommended*

face tomorrow  
SIGGRAPH 2007

## Shader Reflection

- Runtime support to analyze a compiled shader
- ID3D10ShaderReflection interface lets you:
  - Look at your constant buffers and all other constants
  - Look at the parameters to your shader function
  - Look at what resources a shader uses
  - Get semantic bindings on any input/output/constant

face tomorrow



## Compatibility with < SM4

- As a rule, language extensions are emulated when feasible
  - We're still good at removing complexity
- Attributes work across all targets
- The new syntax is a superset of the previous syntax, so SM4 and SM3 shaders can coexist in the same file

## Switching It Up

- Switch statements are now supported in HLSL
  - Maps to a switch instruction in D3D10
  - Emitted as a series of `if` statements in D3D9
  - Supports attributes to decide how it should be emitted
  - Currently does not support non-empty fall-throughs

## Switching It Up

### An Example

```
switch(u)
{
    case 2:
    case 3:
        return 0;
    case 1:
        return 1;
    case 0:
        return 2;
}
```

- Will be emitted using a set of `ifs` that execute a binary search for `u` in {0,1;2;3}

## Switching It Up

### An Example

```
switch(u)
{
    case 1:
    case 3:
        return 0;
    case 2:
        return 1;
    case 0:
        return 2;
}
```

- Won't do a binary search because 1 and 3 would straddle 2, and we'd have to emit it twice



# Switching It Up: Attributes

- Switch statements support 4 attributes
  - **[flatten]**
    - Emit as a series of `if` statements and flatten them
  - **[branch]**
    - Emit as a series of branching `if` statements
  - **[forcecase]**
    - Emit using hardware switch instructions
  - **[call]**
    - Emit using hardware switch instructions where each case is a subroutine

## Sample Method Mappings

- 10 texture types are automatically mapped when they can be to down-level targets
- Works for all sampling types except `SampleCmp` and `SampleCmpLevelZero`
- Down-level targets do not support MSAA textures or texture arrays.
- Down-level samplers are bound to specific textures, so the compiler generates a new sampler in the constant table  
`<SamplerName>+<TextureName>`

face tomorrow



## Integer Types

- All integers are mapped to floats on down-level targets (SM 3.0 and below)
- All integer vectors and matrices are mapped to float vectors and matrices.
- Proper operations are used to maintain integer values in these registers.
- Only exception is in loop optimizations and i registers in SM 3.0 & VS 2.0
  - If an integer is used only as an int loop counter it is only assigned an i register.
  - Otherwise it's bound both an i register and a c register.

SIGGRAPH 2007



## Best Practices

- HLSL does a lot for you
- Compilation Optimization & Performance
- IEEE strictness and configuring the compiler for precision
- A few things will save you some pain when it comes to both 10 and 9
  - Signatures using structs
  - Macros for customization and specialization



## Compilation Optimization & Performance

- Larger shaders can cause slow compilation times
- Highly dependent on the amount of optimization done
- /Od flag disables all optimization – fastest
- /O0 - /O3: The higher the flag number, the more optimizations and the slower the performance
- /O1 is the default

face tomorrow



It's also important to note that we offer the ability to access the original D3D9 compiler for shader models 1.x through 3.x with FXC and D3DX via the /Ld flag. This will load D3DX9\_32.DLL if available and compile the shader with that binary.

Equivalent D3DX flags:

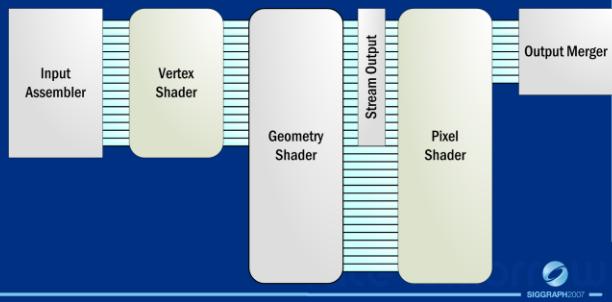
/Od: D3D10\_SHADER\_SKIP\_OPTIMIZATION  
/O0 - /O3: D3D10\_SHADER\_OPTIMIZATION\_LEVEL0 -  
D3D10\_SHADER\_OPTIMIZATION\_LEVEL3

# IEEE Strictness

- Floating point operations are much more strict in D3D10
  - FP32 shader ops – precise to 1.0 ULP
  - FP32 to Integer – precise to 0.6 ULP per op
  - FP16 blending - precise to 0.6 ULP per op
- Using IEEE-strict mode, compiler won't re-order or optimize expressions in ways that change NaN/INF propagation
  - /G1s flag for FXC, D3D10\_SHADER\_IEEE\_STRICTNESS for D3DX10
  - IEEE-compliant min/max that squash NaNs
- snorm keyword – a floating point value is in the range -1 to 1
- unorm keyword – a floating point value is in the range 0 to 1

## Digression: Signatures

- D3D10 API Design Imperative: No draw-time fix-up required
- Fixed, generic register bank connects stages
- Shader linkage enforced via “Signatures”



Interstage linkage works differently in 10.

We've moved away from the “bind-by-name” model in favor of a scheme that can be classified as “bind by position”

The model is as follows – think of it as a bank of registers in between each stage.

Each stage outputs data in a certain order, the next stage consumes it in that order.

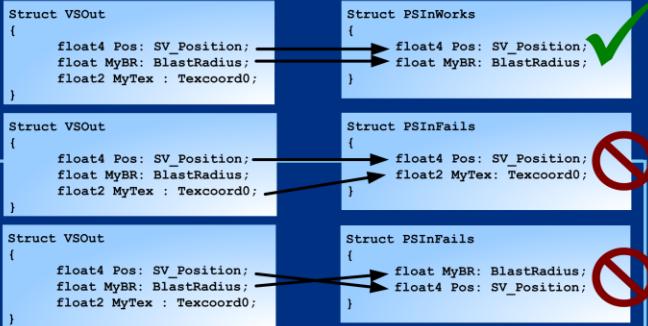
And you bind by position in the register bank – linkage is identified by physical location.

What this means is that instead of doing mapping at draw time, it's up to the application to maintain order at shader author-time.

In other words, we've pushed this work to the outer loop.

And these linkages are maintained and tied to shaders through a construct we call signatures.

# Signatures



- Correspond to shader parameter declarations
  - Aggressively packed by HLSL
- Like a Struct: *Order matters*
- Put “optional” params at the end
- Specialization/packing can save you precious interpolators
- App-defined semantics permitted!

Order matters with inter-stage linkage. Chances are if you’re porting D3D9 shaders to D3D10, this will be a major piece of the porting.

# Structured Signatures

- Use a struct as your only output from a shader stage and your only input into the next

```
struct MyPSIn
{
    float4 pos : SV_Position;
    float4 tex0;
    float4 normal;
};

MyPSIn MyVS( MyVSIn input )
{
    MyPSIn result;
    ...
    return result;
}

Float MyPS( MyPSIn input)
{
    ...
}
```

## Structured Signatures

- Gives you a single definition of your shader signature
- Guarantees the inputs and outputs match
- Sets you up for easy inter-stage specialization using macros:

```
struct MyPSIn
{
    float4 pos : SV_Position;

#ifdef ENABLE_LIGHTING
    float4 normal;
#endif
};
```

## Macros

- Used just like in C/C++
- Additionally, can be set when compiling
  - From FXC: `fxc.exe /D ENABLE_LIGHTING=1`
  - From an app: D3DXMACRO structs and D3D10\_SHADER\_MACRO structs can be passed into all compilation APIs.

# Macros

- Can be used to make a single shader that can be specialized to a number of different configurations
  - Hardware targets
  - Lighting environments
  - Much more!
- Allows you to control the number of inter-stage registers used to optimize the pipeline
  - Easy to do with struct signatures shown above.
- Be careful not to overdo it!
  - Too many axes of specialization can force you to compile hundreds of shaders!
  - Pick the low-hanging fruit to optimize your pipeline

face tomorrow



## Using the D3D10 Compiler

- 2 versions of compiler
- “inbox” compiler provided with Windows Vista
  - Available through `D3D10CompilerShader` and `D3D10CompileEffect` APIs
  - Part of the OS so it cannot be updated
- SDK compiler available via D3DX and FXC included in the DirectX SDK
  - `D3DXCompilerFrom*` functions
  - Updated bi-monthly with the latest compiler optimizations and bug fixes.

## Conclusion

- Use the latest HLSL compiler to take advantage of D3D10 shader features on all platforms
- Be aware of how the compiler maps to down-level targets
- The latest version of the compiler available in the DirectX SDK:

<http://msdn.microsoft.com/directx>

face tomorrow





# Effects 10: Driving the New Effects System

Sam Glassenberg  
Lead Program Manager  
Direct3D Team  
Microsoft Corporation

*Effects 10 (Sam Glassenberg): A review the Direct3D 10 Effects System – a series of APIs to efficiently abstract and manage GPU device state, shaders, and constants. This talk covers the methods to reflect and manage material content as effect (.fx) files. [45 minutes]*

# Presentation Topics

- Effect System Overview
- Design Paradigms
- Walkthrough
  - The Basics
  - Reflection
  - Variables
  - Effect Pools
- Conclusion

face tomorrow



## What is the Effects System?

- Platform for expressing and configuring state for the graphics pipeline
- Includes actual shader source as well so shaders can be included in the pipeline
- Consists of FX file format and FX runtime
- Can be built upon to extend functionality for custom tools



Before we start, I want to cover the basics of the FX system.

The FX system is really a platform for expressing shaders and device state. In other words, you can use the FX system to abstract away some of the complexities of dealing with certain device objects directly. When I say ‘abstract away’, I don’t mean that FX will do all the work for you and that you won’t get to see the device. The nice thing about the FX system is that you can pick and choose which parts of it you want to use, and you can get a varying degree of abstraction.

The layers you don’t end up using will not hinder performance; the fx system is built to scale in terms of functionality, and I will show you that later on in the presentation.

We went through a lot of trouble to separate the content creation aspect and the runtime aspect. Advanced features like the full reflection and annotation layer can be removed at runtime, and what you will end up with is just the runtime data needed to ‘talk to the device’. Good stuff.

## Why Use FX?

- Can treat hardware state and shaders as *content*, not code
  - Update in-place without recompiling your app
  - Use metadata during content creation and development, optimize out for the final build.
- Analyze and inspect effects using reflection
- FX10 is optimized for the D3D10 pipeline
- FX is a standard used by many DCC apps including Softimage, 3D Studio Max, etc.
  - You can start authoring effects before your engine is implemented
  - Use DCC software to get a preview of how your content will look



## Why Use FX?

- FX is Componentized: Pick and choose what parts to utilize when
- Use FX for DCC integration and art authoring prior to engine implementation
  - Use reflection to map the data to your own internal format.
- Use FX for managing hardware state or shaders or constants
- Use FX to handle a subset of materials or data
  - manage all material constants but optimize your critical data updates yourself

face tomorrow



## FX 9 & FX10

- FX10 system has several improvements from the FX9 runtime
- This talk will be focusing on the FX10 system
- Improvements to the FX compiler, now with performance warnings
- Significantly faster and leaner runtime



What's new in FX10:

FX10 is brand new.

It has been rewritten from scratch, both the compiler and the runtime. It now targets D3D10, which means that it has native support for D3D10 hardware concepts.

D3D10 allows for unprecedented performance, but it also may require more resource management than before. FX10 takes care of that. As you will see in a bit, we take care of all resource and dependency tracking.

But the key goal for FX10 is performance. One of the problems with the old FX system is that we had some hidden performance cliffs. You would use the system one way, change something, and all of a sudden you would experience a performance drop. We're putting a stop to that. The FX10 compiler is instrumented to give performance warnings any time you do anything suboptimal. The same holds for the runtime. In debug mode, the runtime can monitor usage and issue warnings when it takes a slow path.

You could ask at this point why we have all these performance cliffs, what are they there for. The reason we have them there is to provide you with more flexibility when prototyping, or to let you perform operations that are more suited to tools. But when you want to get the most performance out of effects, you can always instruct the effect system to drop all reflection and annotation layers.

## FX Runtime

- Performance is critical. O(1) for all common tasks.
- Runtime code is minimal
- Full featured reflection layer can be removed at runtime to reduce memory load

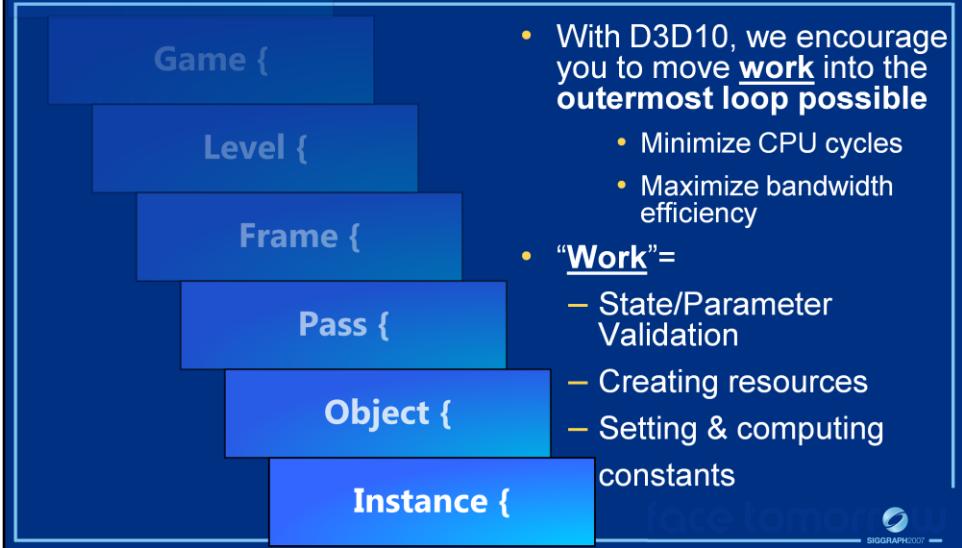
face tomorrow 

## Effect System Scaling

- Built to scale well
  - All Set calls are  $O(1)$
  - All Apply calls are  $O(\text{dependency count})$
  - No  $O(\text{number of objects \& effects})$  operations anywhere in the system
- Performance metrics
- Memory costs

Face to motion  
SIGGRAPH 2007

# Frontload the Work: a D3D10 Paradigm



Now that we've given an in-depth look at how the runtime works, I want to talk about the best practices for using it in your application to maximize performance.

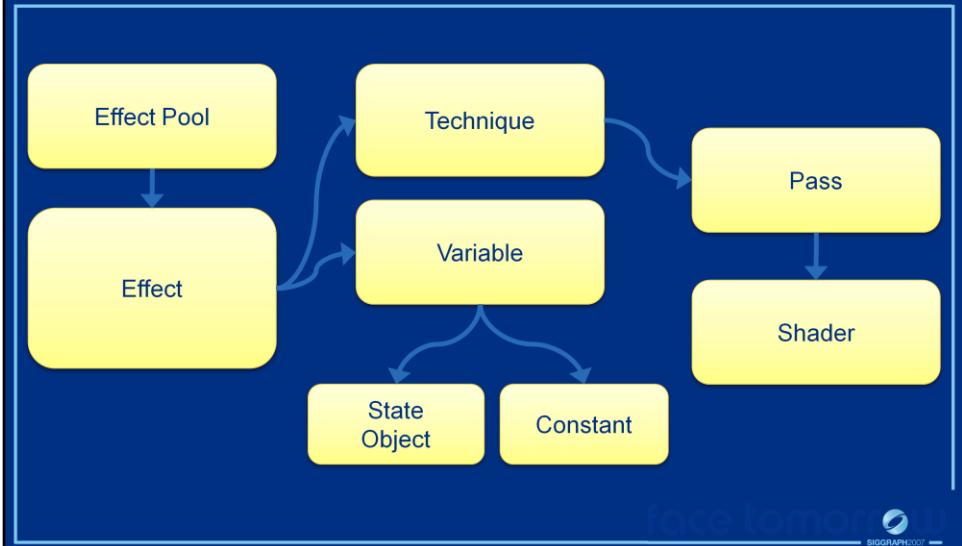
The pipeline is stalled in some way whenever the data needs to be created or updated. By controlling the frequency at which data needs to be fed to the pipeline, we can minimize the overhead of each state update, resource creation, or constant modification

## Frontload the work

- FX10 is designed to also work in this manner
- Reflection system is designed to be used at “Create” time to get interface pointers to all the variables or CBs that need updating
- CBs allow for updating constants at similar frequencies
- Variable setting and Apply() are very fast



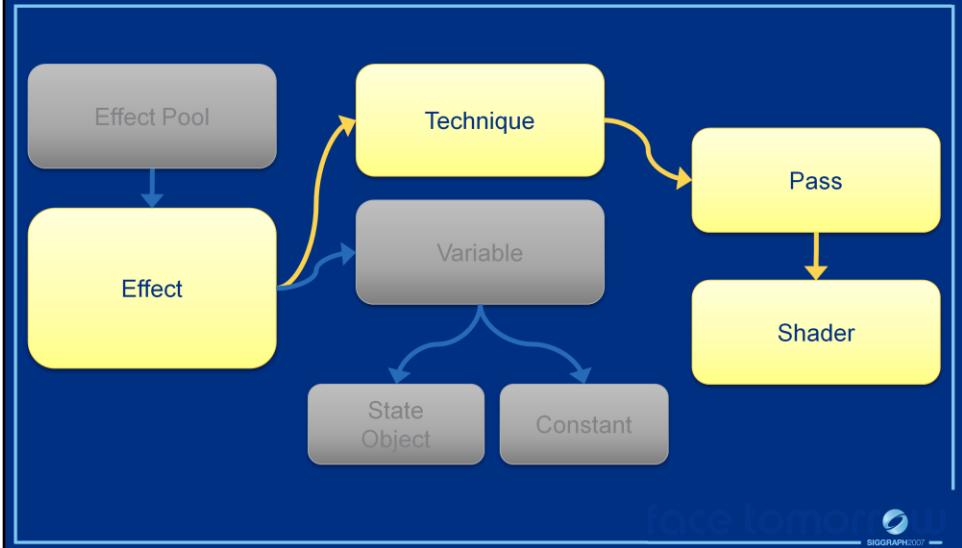
# FX Architecture



face tomorrow



# FX Architecture: The Basics



# The Effect System API

- Creating an effect (ID3D10Effect)

```
hr =D3D10CreateEffectFromFile(
    pFileName,
    pDefines,
    Flags, ...,
    &pEffect,
    &pErrors);
```



## Typical Effect API Usage

- Load an effect (text or binary)
- Query for variable and pass interfaces
- Release reflection data
- At runtime, set variables
- Apply passes

face tomorrow SIGGRAPH 2007

Effect files come in text or binary. Text: good for prototyping, easy to understand what's going on. Binary: great for runtime

The way you use FX is through interfaces (more on this later). After loading a file, query for interfaces.

Release reflection data, this freezes the FX interface, you can't query for things by name anymore, but all the data you don't need at runtime gets dropped. After this point your effect might take up as little as 900 bytes + backing stores for your CBs and state objects

At runtime, use the interfaces to modify values, and apply passes. This is executing the effect.

## Obtaining Effect Interfaces

- FX API is interface based. Not COM, though you can get a lightweight interface to just about anything (variables, techniques, passes, etc.)
- All validation happens when you get the interface.
- No runtime validation



FX API is interface based. Instead of having FX9 style handles, you now get these interfaces which you can operate on. This is a little bit easier because you no longer have to use both an effect interface and a handle, everything is now rolled into one pointer. There are some other advantages to this approach which I will cover in a bit.

## FX Files

- FX files contain:
  - Shader functions
  - Constants used by shaders
  - State objects
  - *Techniques and passes*

face tomorrow



Let's cover FX files.

FX files are the most commonly used part of the FX system. They are a way to describe shaders and state objects.

Since FX is meant to make shader and device state easy, the key parts of FX files are shader functions, device state objects, constant buffers (this is a D3D10 concept, I'll cover that in a bit), and techniques and passes, which are bits of FX code that get executed at runtime.

## A Simple FX file

```
float colorScaleFactor; // Variable declaration
texture2d texLightMap;

float4 PShader(float4 col:Color) : Color // Shader code
{
    return col * colorScaleFactor;
}

technique t0      // Techniques and passes
{
    pass p0
    {
        SetPixelShader(PShader);
        ...
    }
}
```

SIGGRAPH2007

It's a lot simpler to demonstrate these concepts with a sample FX file, so here is what one would look like.

This is a very simple file, it defines a couple of variables that are used by a shader, and a technique and a pass that set this shader to the device. So far this looks just like a regular shader file, with the addition of a technique and pass that perform some extra actions. In this case, you can set a shader.

## Techniques and Passes

- Passes are what really gets executed
- Techniques are used to group passes and organize them into a hierarchy
- You can group passes any way you like, no performance impact

face tomorrow 

SIGGRAPH2007

## Using Techniques and Passes

An FX file contains zero or more techniques, and each technique contains one or more passes.

```
ID3D10EffectTechnique *pTech;
```

```
ID3D10EffectPass *pPass;
```

```
pTech = pEffect->GetTechniqueByIndex(0);
```

```
pPass = pTech->GetPassByIndex(0);
```



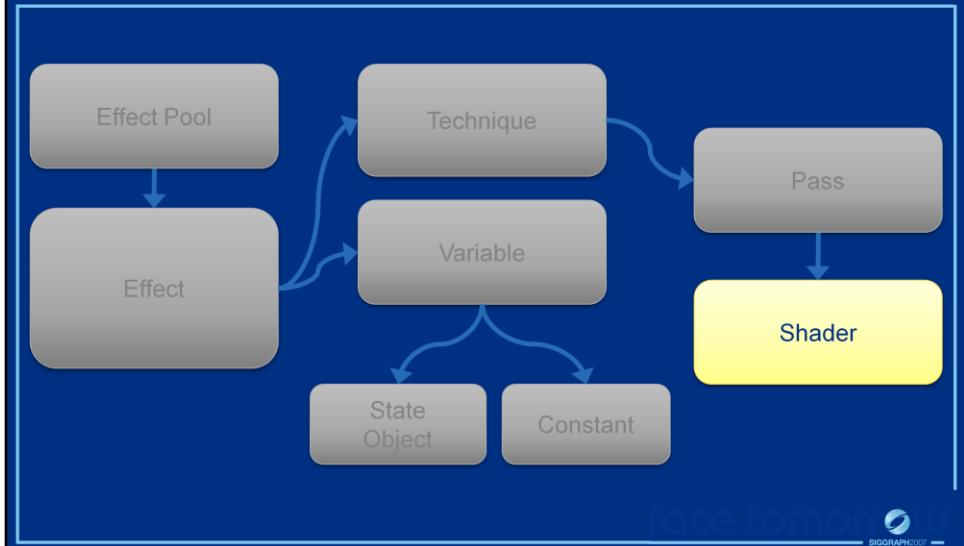
## Applying Passes

- Applying a pass modifies only the state that is referenced in that pass
- With reflection, you can access what state was updated with ComputeStateBlockMask()

```
ID3D10EffectPass *pPass;  
  
pPass->Apply();
```



# FX Architecture: Shaders



face tomorrow SIGGRAPH 2007

## Shaders

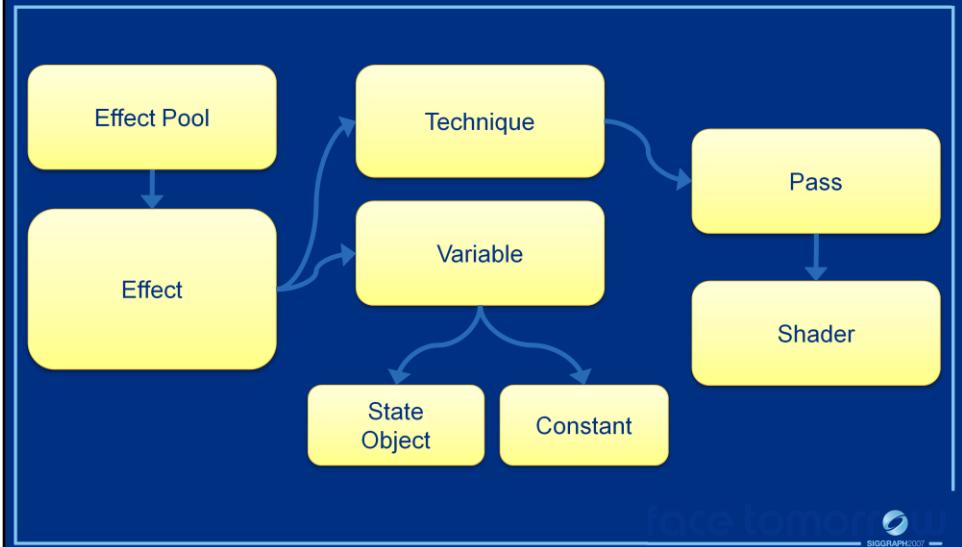
- Compile Shaders in your FX file with `CompileShader` function.
- Set them per-pass using `SetVertexShader`, `SetGeometryShader`, and `SetPixelShader`
  - GeometryShader can be set to NULL

# Shaders

```
technique10 RenderScene {
    pass P0 {
        SetVertexShader(CompileShader(vs_4_0, RenderVS(true)));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, RenderPS(true)));
    }
}
```

- You can set parameters of shader functions to default values in the CompileShader syntax
  - Extremely useful for specialization

# FX Architecture



face tomorrow



## FX Reflection

- You can programmatically inspect an FX file, almost to the point where you can rebuild the file
- This is very important for tools or highly data-driven applications
- But this extra information can be discarded at runtime

face tomorrow 

## How To Use Reflection

```
pEffect->GetDesc(&desc) ;  
  
for (i=0; i<desc.GlobalVariables; i++) {  
    pVar = pEffect->GetVariableByIndex(i) ;  
    pVar->GetDesc(&varDesc) ;  
    pVar->GetType(&varType) ;  
}
```

SIGGRAPH2007

## Things That Can Be Reflected

- Variables (name, semantic, dimensions)
- Variable types, including structs and arrays
- Techniques and passes
- Constant buffers
- State objects
- ... and more!

face tomorrow 

## Extending the Effect System

- Variables, techniques, and passes can be annotated with additional variables that can be reflected back
- Annotations can be discarded at runtime

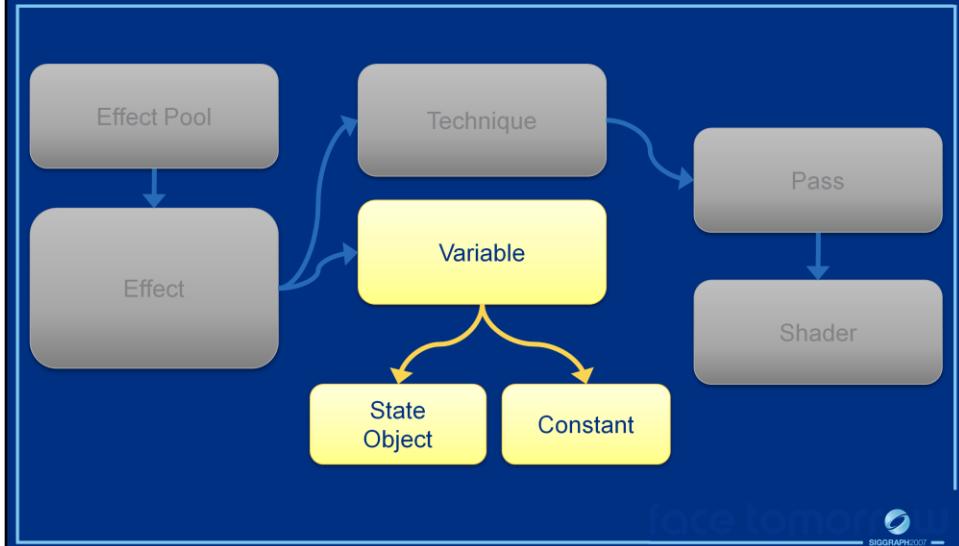
```
float4 lightDir  
<  
    string SasUiLabel = "Sun direction";  
>;
```

## Extending the Effect System

- DirectX Standard Annotations and Semantics define standard ways FX files can be annotated so they interop between different apps
  - Used most commonly with DCC apps like 3dsmax, XSI, Maya
  - You can build your own annotations
  - FX Runtime will help you remove all extra metadata at runtime.



# FX Architecture: Variables



## Variable Interfaces

- Two phase variable reflection
- Generic (untyped) variables
  - Access to type information
  - Access to CB backing store
- Specialized (typed) variables
  - Fast get/sets
  - Specialized access (arrays, structs)

# Variable Interfaces

FX file:

```
float colorScale;
```

Your code (the long way):

```
ID3D10EffectVariable *pVarGeneric;  
ID3D10EffectScalarVariable *pVar;  
  
pVarGeneric = pEffect->GetVariableByName("colorScale");  
pVar = pVarGeneric->AsScalar();
```



# Variable Interfaces

Multiple ways of getting to a variable:

- `GetVariableByName( ... )`
- `GetVariableByIndex( ... )`
- `GetVariableBySemantic( ... )`

Multiple ways of specializing a variable:

- `AsScalar( )`
- `AsVector( )`
- `AsShader( ) ...`



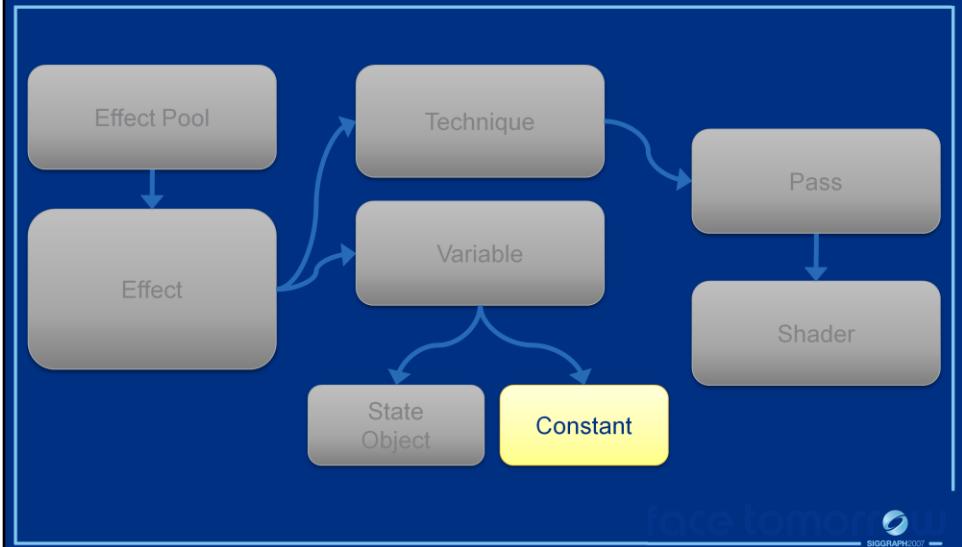
## Getting and Setting Values

- Once you have a fully specialized variable interface, gets and sets are very fast

```
ID3D10EffectScalarVariable *pVar = ...;
```

```
pVar->SetFloat( 10.0f );  
pVar->GetFloat( &fVar );
```

# FX Architecture: Constants



face tomorrow SIGGRAPH2007

# Constant Buffers

- In D3D10 all constants can be placed in constant buffers
- Used to group shader constants together
  - optimizes updating constants from the applications
  - CBs can be cheaply swapped in and out
- Very much like a C struct

face tomorrow



Constant buffers:

Constant buffers are one of the big changes in D3D10, and they need to be reflected in FX10. They are a pretty easy concept to understand: in D3D10, all constants must be placed in CBs. Once you've done that, you can easily swap them in and out. Since any shader that uses a specific CB will access values in that CB using the same layout, if you switch between multiple shaders, you don't need to rearrange all your data in a CB.

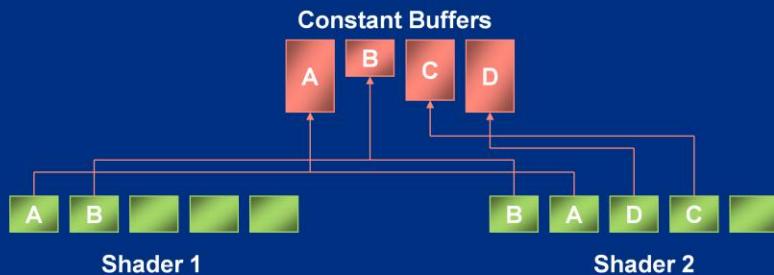
So, back to CB usage:

All constants must be placed in CBs to be used. However, it is up to you how you want to group them. You can put all your constants into just one buffer and not worry about it, but you would probably want to perform some other type of grouping. There are different reasons and ways about how you would group constants, I'll cover that in a bit, and the FX system can help you with that.

So constant buffers are very much like a c struct. All they are is a block of memory, beyond that, to you as the app developer using FX, there is no other magic involved.

# Constant Buffers

- How are constant buffers used by shaders?



This is a bit of D3D10 background information: before we go on, here's how D3D10 shaders use constant buffers. Let's say that we have 4 CBs defined, ABCD. We also have two shaders, 1 and 2. Notice that the two shaders reference the same CBs, but they don't use all of them at the same time, or in the same order. In this case, Shader 1 is using just A&B, while shader 2 is using all 4 but in a completely different order. This is actually a fairly common scenario; let's say that these are vertex shaders that use two vertex buffers (A&B), and the other shader uses a couple of other buffers to get some more work done.

This is where FX10 comes in. With only 4 CBs and two shaders, this isn't particularly hard to do, but once you have hundreds of CBs and hundreds of shaders, keeping track of dependencies can get tricky. We do that for you.

# Constant Buffers

- So what do they look like?

```
cbuffer MyBuffer
{
    float4x4 matWorld;
    float4x4 matView;
    float4    vLight;
    int      arrayIndex;
}
```

This is just a 148 byte block of memory



Here's what a constant buffer would look like in an .fx file. It's very straightforward, you would just use the cbuffer keyword, give the buffer a name, and then list all constants that you would want in that CB.

Keep in mind that this is only a 148 byte block of memory. You can treat it any way you like from your C++ code, but one thing the FX system lets you do with good performance is access these constants using the FX API.

# Constant Buffers

- All FX data constructs still work

```
struct Spotlight
{
    float4x4 matTransform;
    float4 color;
}

cbuffer MyBuffer
{
    Spotlight lightA;
    Spotlight light[5];
    int lightCount;
}
```



You can still use all the higher-level FX concepts with constant buffers. In this particular case, we defined a struct that contains a matrix and a vector, and then we declared a CB that contains an instance of that struct, an array of structs, and a scalar.

By the way, the FX system supports a very flexible set of types; I don't mean just the built in types like float4 vectors or matrices; but also things like 16-bit floats, int matrices, and also compound types, things like structs containing arrays of other structs or whatever. All these types are first class citizens and can be reflected and queried using the high level reflection API.

## Constant Buffers in FX10

- All constants placed outside of a CB get automatically placed in the **\$Globals** buffer
- CBs cannot be nested
- Shaders reference constants in a flat namespace. You can move constants between buffers without having to change shader code

face tomorrow  SIGGRAPH 2007

Flat namespace - might be counterintuitive at first. This is really useful, as it allows you to fine tune your shader by rearranging constants without having to modify shader code. The goal of FX is that you write your HLSL shaders once, your C++ code once, and then you can tune things without having to make these cascading changes to either code base.

## Constant Buffer Usage

- You can declare an unlimited number of CBs
- CBs can have up to 4096 4-channel 32-bit elements
- Each shader can use up to 16 CBs
- Group constants by **frequency of use**
- The compiler will automatically set up shader/CB references for you

Here are some general rules when using CBs.

What you will usually see in most 3D applications is a pattern where certain constants are updated only once (environment load), some other ones are updated in each frame, and certain other ones are updated once per draw call. Since you want to minimize the number of CBs that you need to lock and modify in each pass, it would make sense to group these constants by frequency of use. So you would put all the per-environment constants into one buffer, all per-frame ones into a different buffer, and all others into a separate buffer.

## Advanced CB Usage

- CBs can be explicitly bound to a register slot

```
cbuffer MyBuffer : Register( cb0 )
{
    float4x4 matWorld;
    float4x4 matView;
    float4    vLight;
    int      arrayIndex;
}
```

- D3D10 texture buffers can be used with the tbuffer keyword



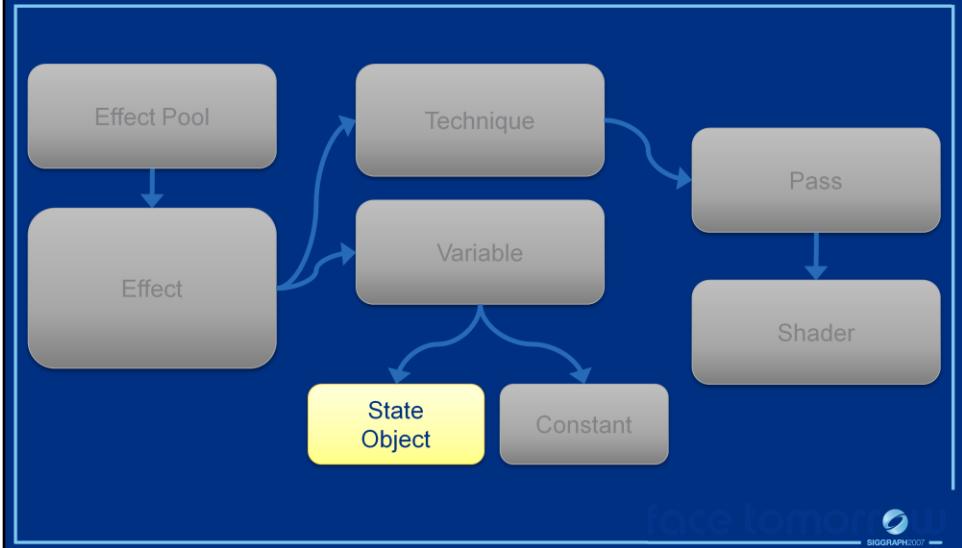
There are a couple of advanced things you can do with CBs. For example you can explicitly bind them to a slot.

We don't anticipate that you'll be using this a lot, but the reason why it's there is if you want to build a hybrid material system, relying on both FX10 and your own system.

As far as FX and HLSL is concerned D3D10 texture buffers work just like constant buffers. TBuffers are a D3D10 concept, and they can give better performance when your data access patterns resemble that of a texture. Switching between a cbuffer and a tbuffer is very simple, just change the keyword. FX10 will automatically create the right type of resource and go down the right path for that type of resource.

This is also an area where we try to help you abstract your code away from the actual hardware details. Note that we're not hiding anything from you, you have full control of whether you're using cbuffers or tbuffers, but switching between them doesn't mean a whole lot of code change on your end.

## FX Architecture: State Objects



face tomorrow



## State Objects

- In D3D10, device state is controlled through state objects
- Four types of state objects; only one object of each type can be active at one time
- State objects are immutable
- There is an upper cap on the number of state objects of the same type you can create at one time (4096)

face tomorrow



SIGGRAPH 2007

# Types of State Objects

- Rasterizer State Object
  - Cull Mode, Multisample Enable, Fill Mode, ...
- DepthStencil State Object
  - Depth Enable, Depth Func, Stencil Masks, ...
- Blend State Object
  - SrcBlend, DestBlend, BlendOp, ...
- Sampler State Object
  - Filter Mode, MinLOD, MaxLOD, ...



Here are the 4 types of state objects.

[Read through categories](#)

Rasterizer, DepthStencil, and Blend get set explicitly in your pass, using a SetState API call.

Samplers are implicit through shaders. In your shader code, if you reference a shader in a texld instruction, we automatically create a dependency. You don't need to do anything else.

## FX State Object Syntax

```
RasterizerState myRS {  
    CullMode = None;  
    SlopeScaledDepthBias = 1.0f;  
    ZClipEnable = false;  
}  
● DepthStencilState, BlendState,  
    SamplerState
```



Here's the syntax for FX10 state objects. There are four types that mirror D3D10 state object types, RasterizerState, DepthStencilState, BlendState and SamplerState.

You specify the type, the name of the state object (this is what you use later on to refer to this object in your pass block), and then list all assignments that you want to set in this state object.

Each state object is a struct, mirrors the D3D10 struct

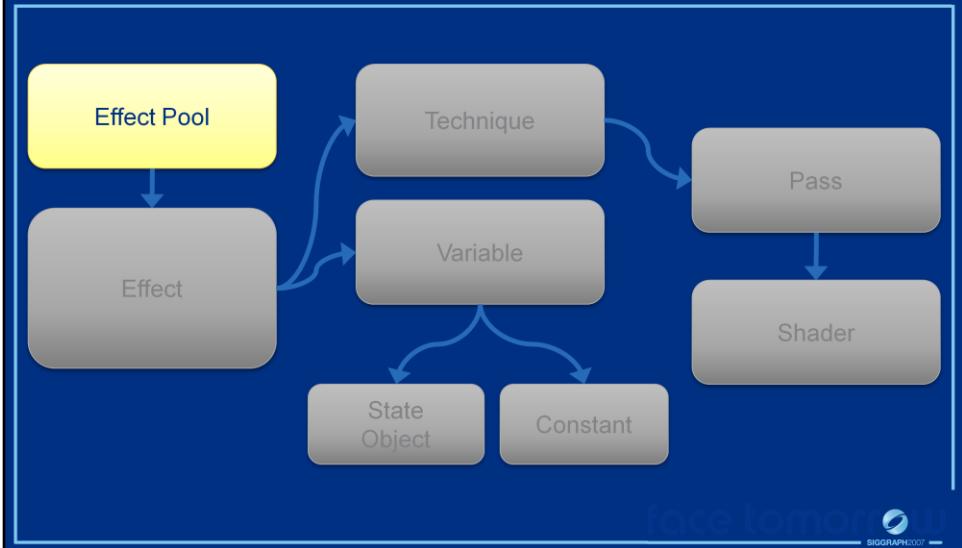
Using FX reflection you can get to this struct

You can only use assignments that are valid in each state object

## Using State Objects

- `SetDepthStencilState( myDS, StencilRef );`
- `SetBlendState( myBS, BlndFctr, Mask );`
- `SetRasterizerState( myRSObject );`
- Switching between multiple objects:
  - `SetBlendState( stateArray[ iBlend ] );`

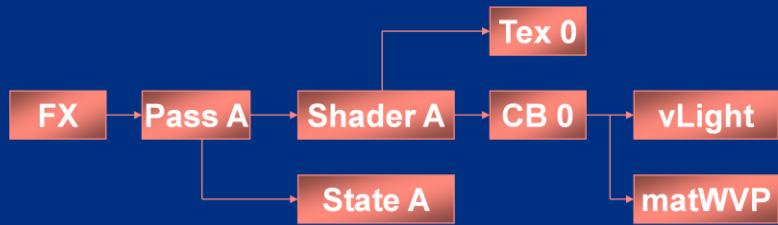
## FX Architecture: Effect Pools



face tomorrow SIGGRAPH 2007

# Runtime Dependency Graph

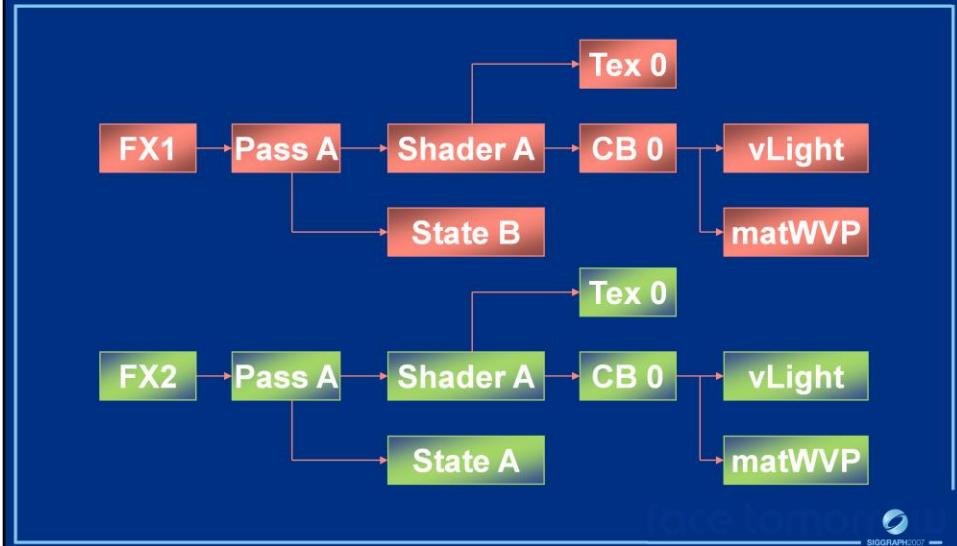
- Runtime keeps track of which dependencies are dirty and updates objects accordingly



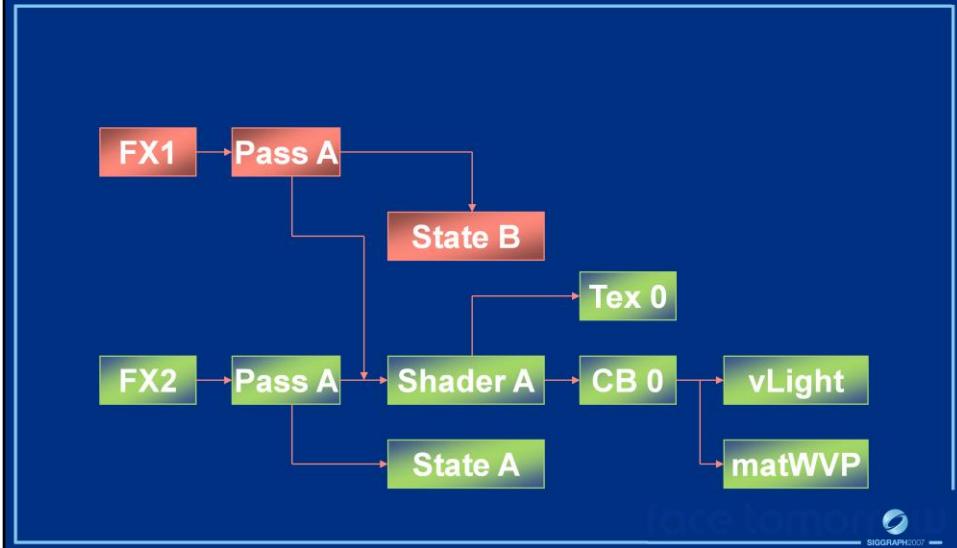
## Effect Pools

- Multiple effects can be loaded into a pool
- Constant Buffers, textures and state objects can be marked as shared
- There is only one instance of each shared object for the entire pool
- You should always use pools, even if you are not sharing any objects right now

## Effect Pool Example



## Effect Pool Example



## Conclusion

- Use the effect system!
- Faster and leaner runtime, but pay attention to perf warnings if you get them
- You can use FX in all stages of development and content creation
- Get the most out of FX by using reflection and SAS. No runtime performance impact.

face tomorrow



Use the effect system. For all it's improvements, D3D10 does add some extra complexity, and FX10 is built to cope with that. You can use parts of FX10, you can use it to set up your state, or maybe just reflect files, or use all of it. I'm confident that the performance and functionality improvements will convince you to use FX10.





*Porting Game Engines to Direct3D 10:* Direct3D 10 is a major revision of the Direct3D API, and poses unique challenges to games supporting multiple platforms while still taking advantage of the new capabilities. This presentation covers practical experiences in updating game engine technology to support and exploit Direct3D 10, and the implications for hosting both Direct3D 9 and Direct3D 10 within the same executable.

Crytek's Crysis presented by Carsten Wenzel [25 minutes]

## Overview

- Introduction
- D3D10 & CryEngine2
- Porting, integration issues and optimizations
- Conclusion

face tomorrow  SIGGRAPH2007

## Introduction

- Fairly early adaptation
- Spec reviews / discussions started in 2005
- Actual work on D3D10 render began middle 2006
- Support from MS and IHVs was crucial to get something running in a reasonable amount of time

face tomorrow   
SIGGRAPH 2007

When we started out nearly everything was early alpha or not even there yet...

- Vista was pre-alpha, DX10 / DXGI still subject of active development
- Early DX10 developer hardware just started to become available (used RefRast only to confirm driver bugs)
- Driver in very early stage, performance analyzer tools not available from the start

## How to support & exploit D3D10 in the context of a multi-platform engine

- D3D10 API redesign offers a more concise view on how modern graphics hardware works
- For the sake of performance it “offloads” a lot of responsibility to the API user



A lot of what the runtime and driver conveniently verified and ensured is now up to the user to implement.

Advantage:

- Get it right at the high level -> better performance due to simplified lower level code
- More transparency and control

But this also means more work and a firm grasp of how individual pieces in the pipeline are connected and work together.

## How to support & exploit D3D10 in the context of a multi-platform engine

- CryEngine2 is designed to be multi-platform
- Render implementations for various platforms / APIs share one interface and lot of common code
  - State & resource management
  - Draw list processing
  - Shader subsystem, etc.

## How to support & exploit D3D10 in the context of a multi-platform engine

- Therefore certain API changes required special considerations
  - State objects
  - Shader model 4.0
  - Constant buffers
  - Strict API validation (shader linkage, etc.)
  - Buffer updates
- Support for new features like geometry shaders, stream out or texture arrays was fairly simple to add



## Porting and making it work

- Render code already designed to separate platform / API dependent parts from independent parts
- D3D10 render implementation is specialization of base renderer
- Inherits interface and all common code

## Porting and making it work

- Platform / API dependent parts wrapped in functions which get called from common code
- Provided stubs and implemented one at a time
- If certain API characteristics were awkward to map to common code flow used `#ifdefs` to specialize behavior



## Special considerations: State objects

- Common render code relies on fine grained control over render states
- As other platform / API dependent code state manipulation is wrapped
- Internally cache state objects (depth, raster, blend, etc.)
- When fine grained state is to be set...
  - Check which category it falls into
  - Get current state object and its description
  - Manipulate description as requested and hash it, ...



Fine grained render state manipulation is required for various reasons:

- Was supported since the very beginning
- Other APIs allow it
- It's convenient to use and expose to client code (client code shouldn't care about state blocks or objects)

Various points in the engine allow manipulating / overriding state:

- Client code
- Shader system
- Renderer might adjust current setup when rendering certain features on lower spec machines

## Special considerations: State objects

- Use hash to look for existing state object matching that description
- Reuse or create it if non-existent
- Finally set new state object
- So far we're building states on demand
- Pre-warming the state cache might be worthwhile though

## Special considerations: Shader model 4.0

- Use our own shader / effect system
- Preprocessor generates HLSL compilable code from shader scripts containing engine specific annotations and semantics
- Updated to support GS, map constants to CBs, etc.
- Use APIs shader reflection interfaces to query required bindings / inputs for later setup
- Existing shaders are compiled in legacy mode. This way they can be used for both DX9 and DX10.
- Enhanced syntax used for DX10 only shaders (e.g. templated samplers to read from MSAA texture)

face tomorrow



## Special considerations: Constant buffers

- Group constants by frequency of update
  - Material, per frame, light info, skinning data, per batch (minimize!)
- Each group maps to a unique CB index
- Constants are bound via a semantic
- For DX10 this semantic also describes the group a constant falls into
- Allows preprocessor to move constants into their appropriate CB scope
  - Existing shaders automatically benefit from efficient binding
  - Source shader scripts devoid of API specific annotations
  - Shader authors don't have to worry about API

For the initial port there was just one constant buffer (CB) that was updated before each draw call. It was convenient to implement (closest to existing DX9 behavior) but far from being efficient. In heavy scenes in-game profiler reported ~7000 updates sending 3-4 MB of constant data each frame.

Wanted a solution that facilitates CBs in a way they are “meant to be used” but keep this transparent as much as possible.

Basic idea was to group constants by frequency of update where each group maps to a unique CB index. A shader constant is bound to the engine via a semantic so its current values can be pulled whenever necessary. For DX10 this semantic also describes the group a constant falls into.

This allows the preprocessor to move each constant into its appropriate CB# scope [by putting “`cbuffer groupName = register(b#) { ... }`” around the actual declaration of the constant].

Advantages:

- Existing shaders will benefit from efficient constant binding automatically
- Source shader scripts are devoid of API specific annotations
- Shader authors don't have to worry about API details when writing shaders

After grouping was implemented the number of updates per frame drop to ~5000. Still high and coincides with the number of draw calls. Most of the updates fall into “per batch” so minimizing/eliminating these as much as possible can gain more performance! So shaders should be checked to see with constants can be either moved to less frequently updated groups or removed all together.

## Special considerations: Strict API validation

- API now much more rigorously checks how user drives it
- Allows for simplifications at lower API and driver levels → less expensive draw calls
- Example: shader linkage
  - Vertex layout / output signatures has to match the expected input signatures of following shader
  - Dimension of each passed element has to match dimension expected in following shader
  - Took some time to clean up / unify signatures
  - Nice side effect: signature-fixed shaders run faster on DX9



API now much more rigorously checks how user drives it. Debug runtime sends detailed error reports. Release runtime will simply discard invalid calls. At times it may seem as it's painstakingly going into every single little detail but it for good reason (performance).

Stricter validation allows for simplifications at lower API and driver levels. For the user this means reduced draw call costs (can either render more detailed scenes at the same time or render faster).

Example: shader linkage

1. Vertex layout / output signatures has to match the expected input signatures of following shader...
  - It either has to be a perfect match or what is read is subset of what gets passed.
  - The order of elements cannot be changed. In DX9 driver intervenes to ensure 1:1 mapping by patching the shader if necessary (which can be expensive).
2. Dimension of each passed element has to match dimension expected in following shader (cannot send a float2 and expect an auto expanded float4) .

It took some time to clean up shader signatures. Since already existing input layouts get reused (don't create input layout for every shader combination!) problems mentioned above occasionally showed up. This can easily happen when several people write shaders. Especially since at the time most shader development was done on DX9 only and wasn't always immediately cross checked with DX10 (DX9 is tolerant to this kind of "sloppiness"). Defined common inputs and reuse them as much as possible to reduce these type of errors.

Nice side effect: signature-fixed shaders run faster on DX9 as driver no longer needs to perform shader patching.

## Special considerations: Buffer updates

- Update methods for truly static / dynamic buffers remain as in D3D9
- However, occasional updates of static buffers can cause severe pipeline stalls (hitches), e.g. updating terrain
- Use staging resources as intermediate storage
- This way mesh update can be pipelined



Update method for static buffers: Upload at buffer creation time

Update method for dynamic buffer: Use dynamic buffers and Map / Unmap with Discard / NoOverwrite flag whenever necessary

Certain types of vertex data that doesn't justify a dynamic buffer as it changes to infrequently (i.e. not every frame). Then static buffers are usually the best choice because they reside in local video memory guaranteeing the fastest render performance. However, if they do need to be updated the following can be done in order to prevent stalls...

- Create pool of staging resources
- Cycle through pool and upload data to the one least recently used
- Use that one to update the actual static buffer (initiated via `CopySubresourceRegion()`)
- Since the static buffer is no longer directly locked by the user, the pipeline doesn't need to get flushed (as buffer might currently be used by the GPU for rendering)

Basically, it imitates behavior of truly dynamic buffers at the user level (buffer allocation, cycling, renaming to prevent pipeline stalls)

## Conclusion

- New D3D10 features are great and perf improvements are definitely there
- When porting over they don't come for free though!
- Expect several refactoring steps in your existing render code to be required in order to drive the API in the most efficient way







*Porting Game Engines to Direct3D 10:* Direct3D 10 is a major revision of the Direct3D API, and poses unique challenges to games supporting multiple platforms while still taking advantage of the new capabilities. This presentation covers practical experiences in updating game engine technology to support and exploit Direct3D 10, and the implications for hosting both Direct3D 9 and Direct3D 10 within the same executable.

*Relic Entertainment's Company of Heroes* presented by Daniel Barrero [25 minutes]

## Requirements for the port

- Graphics should look better or at least identical to the highest quality D3D9 version.
- Changes to existing art assets to support D3D10 features should be avoided as much as possible.
- New D3D10 only features should show clear graphics or performance improvements.

face tomorrow  SIGGRAPH2007

Graphics looking better or at least identical to the dx9 version meant that all existing D3D9 features had to be supported somehow on the D3D10 port.

Given that the game was already finished meaning that we had already tons of graphics assets, all new D3D10 features were limited to things that could exploit the existing assets, that could be generated automatically or that would require only new art without affecting the existing ones.

## What needed to be done

- Update the graphics driver abstraction layer so it can dynamically load different driver implementations.
- Write the D3D10 graphics driver
  - Use the D3D9 driver as reference point.
- Update the content pipeline tools to support the new D3D10 capabilities.
- Implement new features that take advantage of the new D3D10 capabilities.

## Biggest Changes between the D3D9 and D3D10 driver

- Device creation and handling.
- State management.
- Shader model 4.0 changes.
- Texture management and shader resource binding.
- Error checking and things that now have to be done at creation instead of rendering time.
- Render target management.
- No more BGRA support.



Having a platform independent abstraction layer for the graphics driver beforehand helped accelerate the porting process initially, the main problem was that the abstraction layer reflected the previous generation architecture (D3D9/ogl) a bit too close at some places. Meaning that some functionality that doesn't exist anymore in D3D10 needed to be simulated.

Fortunately we didn't use much of the fixed function pipeline, which simplified the port, on the other hand, the parts that we did use like alpha testing required quite a bit of planning to get it working for all our shaders in the same way as D3D9.

Device creation and handling is a lot simpler on D3D10 due to the fixed spec and minimal required feature set, not having to check for cap bits and a lot of special cases simplified the device driver also as all functionality used is guaranteed to be supported.

# State Management

- Has to handle all unique state objects database.
  - Keep track of resources associated to each state object.
  - Handle overriding of single values within a state object.
    - Costly, most of them were replaced by using static branching and adding specific shader techniques with the desire state for the overrides.
- Deals with sampler states.
- Manages mapping of resources to the shaders.

face tomorrow



On dx9 the effects state manager used to handle only render states:

Blending, depth testing, stencil etc.

On D3D10 it handles the state objects and makes sure that all values are set and calculate the differential values to create a new state object if necessary so the final state can be set up atomically. This required being careful when changing states, the whole state is changed not only one value, if only one value is changed then all the others will get a default device value instead of keeping the previous state values.

This means that It also had to handle overriding of single values within a state object. Easy for the stencil ref, more work necessary for things like the stencil mask. Fortunately we handled most of this overrides with static branching on the shaders.

# Shader Pipeline

- Custom D3D10 Shader compiler:
  - D3D effects interface to find information about techniques and passes.
  - Reflection interface to find information about resources and constant buffers bindings.
    - Great for gathering information about the shader off line.
    - Potentially slow to be used in game.
    - Requires having around the shader binary blobs increasing memory usage.

face tomorrow



We only use the fx interface for our shaders off line as containers to be preprocessed. That meant that we needed to create a new custom shader compiler for D3D10, for that purpose we had to rely heavily on the reflection interface, to get specific information about constant buffers and resource bindings.

Given that we support all kinds of shader models, we already had in place a system to handle multiple versions of the shaders from the same code base, having a separate D3D10 compiler also allowed us to keep supporting older shader models.

## Shader Pipeline

- Use an include file with compatibility macro definitions instead of the backwards compatibility flag:
  - Keep sharing the code base between shader models.
  - Allow compiler to do full optimization.
  - Easier handling of semantics naming convention differences between D3D9 and D3D10.
  - Differences on state handling are managed on a single file.
  - Component swizzling due to BGRA to RGBA support is concentrated in one place.



Our shader codebase is shared among all the shader models that we support : 1.1, 2.0, 3.0 and 4.0, to be able to reuse code and get the most out of the shader compiler instead of using the compiler compatibility flag we use the preprocessor to define compatibility macros allowing us to manage semantic changes more easily. Also this means that changes other than specific shader models features are contained in a single file.

Since all of our graphics assets were created with dx9 optimizations in mind, we needed a way to handle subtle differences between D3D9 and D3D10 like the handle of color formats as to reduce memory we compress vertex and uv coordinates information into color formats instead of passing them directly as floats. This meant having to convert the bgra input to rgba when necessary.

## Shader Pipeline

- Support for geometry shaders to the shader compiler and technique generation scripts.
- All scripts needed to be changed to gather the information of all states for a given technique so to generate unique states that can be applied atomically.
- Shader signatures strict matching policies required reorganizing the shaders inputs to better match all the possible combinations.
  - Shader resources and constant buffers are bound now per pass and per specific shader (VS/GS/PS) and could be different for each one of them.

## Textures

- Texel centering has to be adjusted dynamically based on the device settings.
- Sampler states have to be managed by the application now.
- Staging textures have to be managed carefully otherwise will cause memory fragmentation problems.
  - Reuse them as much as possible.
  - Use UpdateSubResource instead whenever possible.



Texel centering was off not only for screen space objects like the UI, postprocessing and fonts, but also for a lot of textures as we pack smaller into texture atlases to avoid memory fragmentation and lessen the cost of texture resource swapping on the graphics card. This meant that the device had to expose the texel center offset and all the places that used it needed to be updated for that purpose. This also caused very visible artefacts with antialiasing in the UI and fonts.

With dx9 the sampler states are baked into the shader to the right texture and texture stage, on dx10 as samplers can be reused it required the effects manager to handle them independently and it also had to make sure that they were bound to the right places when required by the texture resource on each shader.

## Vertex/Index Streams

- BGRA vs RGBA caused problems with some vertex data packed off line as D3DCOLORs.
- Vertex declarations for a single stream have to match the input signatures order.
  - They change per technique/per pass, while on D3D9 per technique was enough.
- Things have to be done at creation time instead of run time:
  - Instanced data vertex stream setup is done at creation time.
  - Error checking.

face to node



BGRA vs RGBA cause problems with texture uv's and vertex data as shaders could receive compressed and uncompressed data at any time, vertex data had enough information to know when a swizzle was needed but uv, had some extreme cases that took us a long time to track and the only way to solve them was doing the format change at loading time.

## Memory

- Frequent creation and destruction of resources at load time caused crashes due to memory fragmentation, to reduce it:
  - Reuse staging textures
  - Flush more often.
- Render targets are now mapped into the application VM Space.
- D3D10GetInputSignatureBlob is your friend.
- Be careful with keeping in sync texture interfaces and shader resources reference counts.



For vertex declarations the shader binary blob is needed to be able to create the matching Input assembly, as we wouldn't know (nor wanted) to create every possible combination as not all of them are used, we needed to keep around the shader binary blobs even after the shader interface was created already on the graphics card, this can add a lot of memory, keeping only the binary part of the vertex shader using d3d10getinputsigntureblob saves tons of memory, still, it takes more than dx9 for this as we could throw away the whole thing out as the driver would use the semantics to map vertex stream and signatures to vertex shader signatures.

## Adding New D3D10 Specific Features

- Alpha to coverage
- Per Pixel lighting
- Omni shadows (GS / Instancing)
- Short grass (GS / Instancing )
- Litter objects (Instancing)

face tomorrow



SIGGRAPH2007

Once the dx10 driver was up and running it was finally possible to start implementing specific dx10 features, the most visible features while running our game on dx10 would be these.

These features were chosen because they were the ones that added the most impact for our type of game. On Dx10 some of them can be implemented in multiple ways, and actually we did try both and stuck with the implementation that had the best performance on every vendor's Dx10 graphics HW.

## Alpha to Coverage

- Small change to the shaders.
- Stipple transparency effect.
- Mostly should only be used with MSAA on.

face tomorrow  SIGGRAPH 2007

Image to be added as soon as we

## Per-Pixel Lighting



D3D9- Mixed (VS/PS)



D3D10 100% Per-Pixel

Face to Face



## Per-Pixel Lighting



D3D9- Mixed (VS/PS)



D3D10 100% Per-Pixel



## Per-Pixel Lighting



D3D9-Mixed (VS/PS)



D3D10 100% Per-Pixel



## Per-Pixel Lighting - Off



SIGGRAPH 2007

## Per-Pixel Lighting - On + Omni Shadows



SIGGRAPH2007

## Omni Shadows/Point light shadows



## Short Grass



D3D10-Grass Off

D3D10-Grass On

SIGGRAPH 2007

## Short Grass - Off



## Short Grass – On



## Short Grass – On + Litter Objects



## Litter objects



Face to face  
SIGGRAPH 2007

## Litter objects



D3D10 No instanced objects

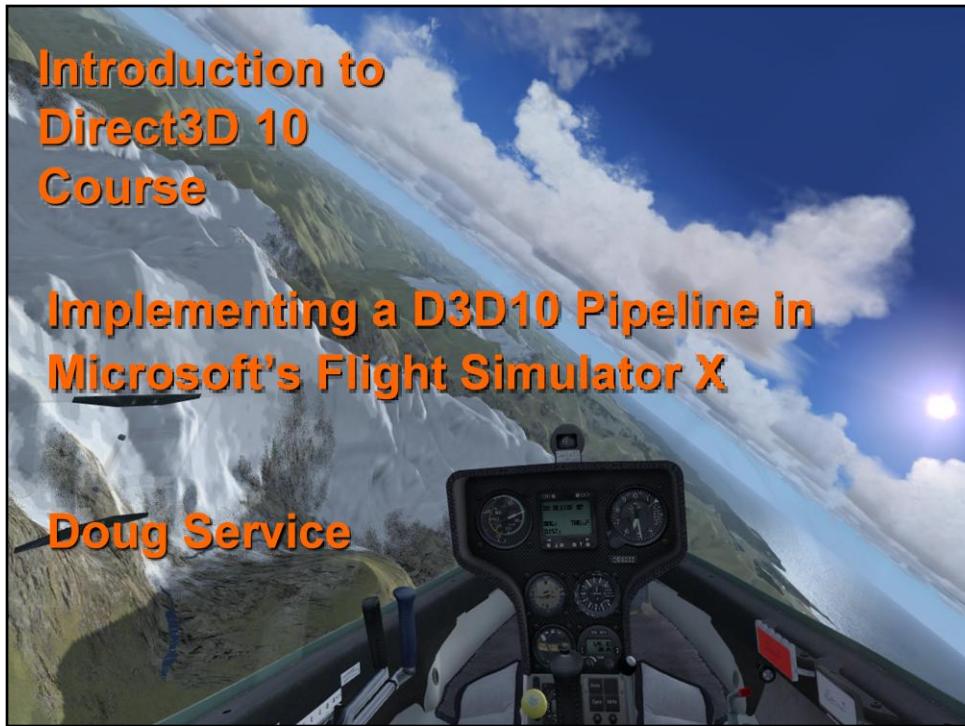
D3D10 Instanced litter objects



## Conclusions

- D3D10 is a lot cleaner than D3D9
- Different enough from D3D9 that it took a lot more work than expected.
- If instancing can be used instead of the GS go with the Instancing.





*Porting Game Engines to Direct3D 10:* Direct3D 10 is a major revision of the Direct3D API, and poses unique challenges to games supporting multiple platforms while still taking advantage of the new capabilities. This presentation covers practical experiences in updating game engine technology to support and exploit Direct3D 10, and the implications for hosting both Direct3D 9 and Direct3D 10 within the same executable.

*Microsoft's Flight Simulator X by Doug Service [25 minutes]*

## Overview

- Unique aspects of Flight Simulator X (FSX)
- D3D9
  - Adapter enumeration
  - Devices, swap chains, and texture sharing
- D3D10
  - Adapter/output enumeration
  - Devices, swap chains, and texture sharing
  - Multi-monitor setup and rendering pseudo code
- Questions

face tomorrow



## Unique Aspects of FSX

- Large variety in end user demographics
  - Serious pilots want simulator training value
  - Casual gamers want high entertainment value
  - Various combinations between the two extremes
- Multiple windows on multiple displays in both full screen and windowed mode.
- Users can drag windows across displays in both full screen and window mode



# Unique Aspects of FSX

## Dual Monitor Window Mode



face to motion



# Unique Aspects of FSX

## Dual Monitor Full Screen Mode



SIGGRAPH  
SIGGRAPH2007

## Unique Aspects of FSX

### High End User Setup with Twelve Monitors



SIGGRAPH2007

## Unique Aspects of FSX (Cont)

- Render the entire world from data contained on two DVDs
  - Terrain geometry is captured from public domain and commercial sources
  - Terrain textures in well known locations are generated from photo reference

## Unique Aspects of FSX (Cont)

- Remaining terrain textures automatically generated in real time
  - One kilometer land mass data
  - Date (season) and time of day
  - ~7500 land mass source textures
  - Artist generated masks
  - Vector data for roads, bodies of water, power lines etc

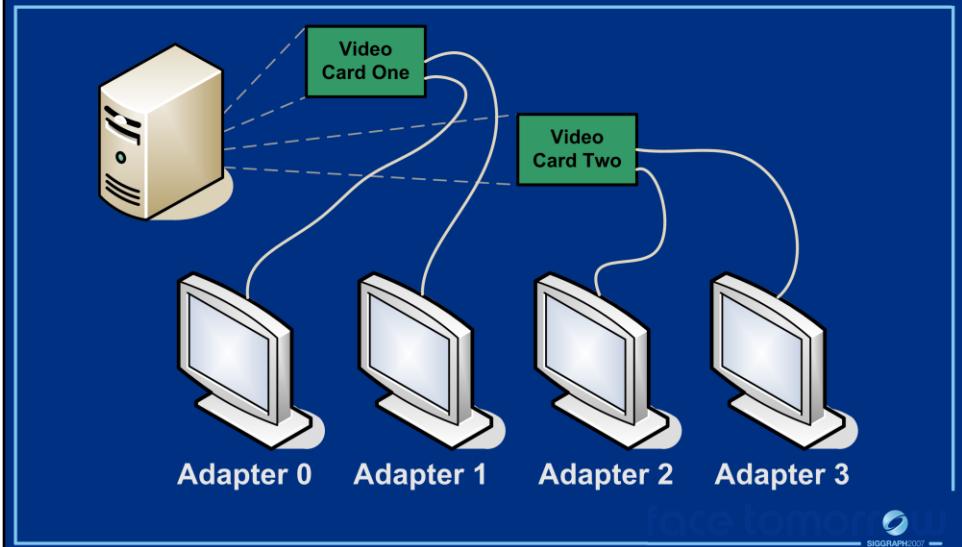
## Unique Aspects of FSX (Cont)

- Well known buildings and terrain landmarks are modeled by artists
- Vegetation and remaining buildings are automatically placed using land mass data and artist generated masks

## D3D9 Adapter Enumeration

- Enumerate one adapter per monitor
  - IDirect3D9::GetAdapterCount()
  - IDirect3D9::GetAdapterIdentifier()
    - Returns a D3DADAPTER\_IDENTIFIER9 structure
- Dual monitor video card enumerates as two adapters

## D3D9 Adapter Enumeration

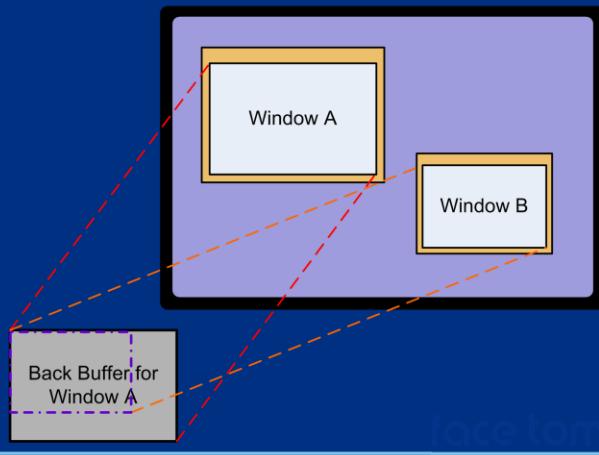


## D3D9 Devices and Swap Chains

- Swap chain created with device and associated with the window handle
  - IDirect3D9::CreateDevice()
- Buffer flip can be redirected to a different override window when present is called
  - IDirect3DDevice9::Present()

## D3D9 Swap Chains

Can redirect from Window A's back buffer to Window B by passing Window B's handle to IDirect3DDevice9::Present()



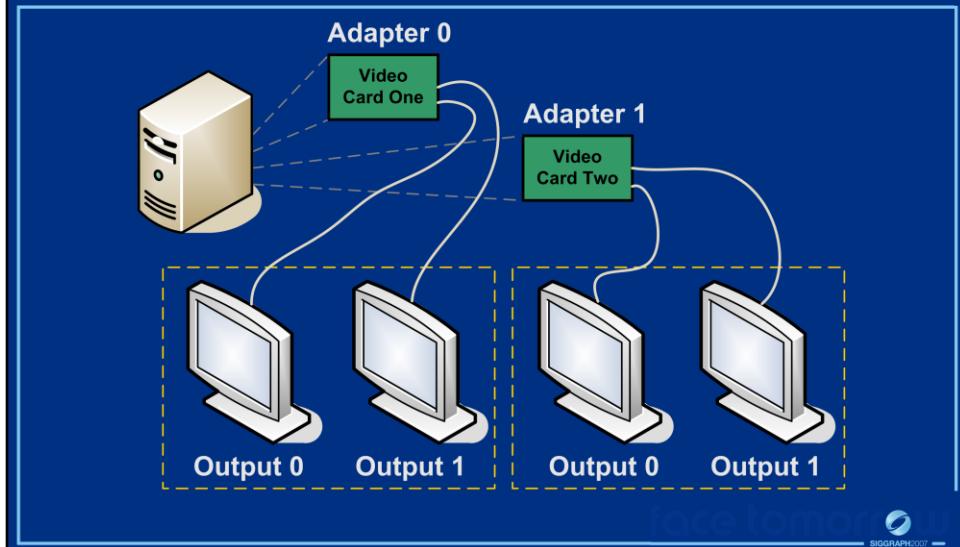
## D3D9 Devices and Texture Sharing

- Textures cannot be shared between devices
- Have to upload a texture twice on a dual monitor card even though both adapters share the same video memory
- Cuts the amount of texture memory in half if all textures are used on both adapters.

## D3D10 Adapter/Output Enumeration

- Enumerate one adapter per video card and one output per monitor
  - IDXGIFactory::EnumAdapters()
  - IDXGIAdapter::EnumOutputs()
- Dual monitor video card looks like one adapter with two outputs

## D3D10 Adapter/Output Enumeration



# D3D10 Adapter/Output Classes



face tomorrow



SIGGRAPH 2007

## D3D10 Devices

- Device is bound to the adapter and is created independent of the swap chain
  - D3D10CreateDevice()
- Device can render to all swap chains it is associated with on the adapter
  - Implies device can render to multiple adapter outputs

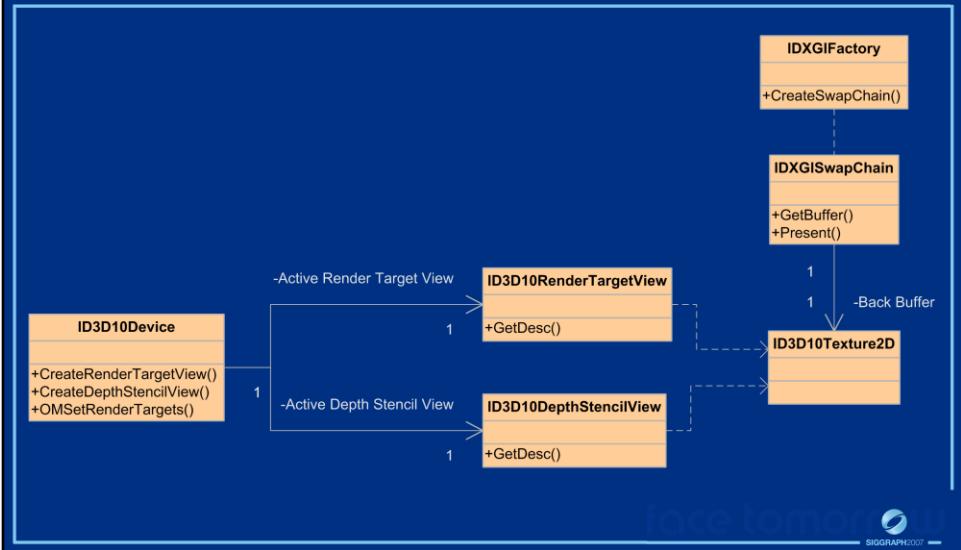
## D3D10 Swap Chains

- Swap chain is bound to the device (adapter)  
IDXGIFactory::CreateSwapChain()
- Swap chain becomes active output for a device through a render target and depth stencil view
  - ID3D10Device::CreateRenderTargetView()
  - ID3D10Device::CreateDepthStencilView()
  - ID3D10Device::OMSetRenderTargets()
- Swap chains are bound to a window handle and a buffer flip cannot be redirected to another window
  - IDXGISwapChain::Present()

face tomorrow



# D3D10 Swap Chains



face tomorrow



SIGGRAPH 2007

## D3D10 Devices and Texture Sharing

- Textures can be shared between devices associated with the same adapter
  - ID3D10Device::CreateTextureXD(), X = 1,2,3
    - Set D3D10\_RESOURCE\_MISC\_SHARED flag in MiscFlags field of D3D10\_TEXTUREXD\_DESC

## D3D10 Multi-Monitor Setup

```
CreateDXGIFactory()  
IDXGIFactory::EnumAdapters()  
IDXGIAdapter::EnumOutputs()  
for each output  
    IDXGIOutput::GetDisplayModeList()  
for each adapter  
    D3D10CreateDevice() // Device can draw to all adapter outputs  
for each window  
    DXGIFactory::CreateSwapChain() // Full screen: 1 window per output  
    IDXGISwapChain::GetBuffer()  
    ID3D10Device::CreateRenderTargetView()  
    ID3D10Device::CreateTexture2D()  
    ID3D10Device::CreateDepthStencilView()
```



## Multi-Monitor Rendering

simulate world

for each device (adapter)

ID3D10Device::OMSetRenderTargets()

ID3D10Device::ClearRenderTargetView()

ID3D10Device::ClearDepthStencilView()

...

Draw

...

IDXGISwapChain::Present()



## Questions?

- See DirectX SDK MultiMon10 sample
- [www.microsoft.com/directx](http://www.microsoft.com/directx) for latest DirectX SDK and documentation

face tomorrow  SIGGRAPH 2007



# **Content Tools & Film Use of Direct3D 10**

Nick Porcino

LucasArts



*Content Tools and Film Use of Direct3D 10* (Nick Porcino): The Direct3D 10 pipeline offers unique opportunities for producing high-end graphical content, but requires a new generation of tool capabilities. With the new leap in consumer-level Direct3D hardware capabilities, interest in film pre-visualization and GPU-based acceleration continues to grow. [30 minutes]

## Introduction

- Presentation by Nick Porcino
- Opening reel: ILM & Lucasarts

face tomorrow   
SIGGRAPH 2007

## Blinn's Law

- All frames take 45 minutes!
- The 45 minute frame is going away

face tomorrow



Recent discussions of high end rendering have quoted Blinn's Law as “all frames take 45 minutes.”

This homily can be translated “as hardware and software improve, we demand proportionately more of them.”

The 45 minute frame is going away. We leverage coherency and caching to move that time elsewhere in the pipeline, or to reuse it so that the 45 minutes applies to many frames.

## Caching and Coherence

- Content pipelines are like game loading – precompute and cache!
- Caching and coherency is the basis of an efficient content pipeline

face tomorrow



SIGGRAPH 2007

Game loading as analogy - the slow work has been precomputed and cached offline in order to support real time rendering in game. This notion should be considered carefully in pipeline design to reduce work occurring everywhere in the pipeline.

Cached data are the intermediate results to be used by later processing stages. The notion of caching leads directly to the existence of pipelines.

Coherency refers to staging resources in the pipeline so that intermediate results can be reused many times with a minimum of reloading.

## Kinds of Pipelines

- Ad hoc
- Structured
- Modular

## Type1: Ad hoc

- Built as the need arises.
- Extremely flexible due to its lack of structure
- By the end of the production, the ad hoc process becomes a liability

face tomorrow

SIGGRAPH2007

An ad hoc pipeline is one that is built as the need arises.

The ad hoc pipeline services the entire product at the same time.

Asset management is typically accomplished via brute force or heuristic methods.

Data formats are arbitrary, and dictated by application need and programmer preference.

The ad hoc pipeline is extremely flexible due to its lack of structure, and supports experimentation and iteration through the application of large amounts of elbow grease and creativity throughout the production process.

This lack of structure becomes more and more expensive as the end of the production approaches.

By the end of the production, the ad hoc process becomes a liability as hacks, workarounds, exceptions, and failing pipeline elements become more common.

## Type 2: Structured

- Sign offs, checks, choke points, and automation.
- The pipeline can be managed by non technical personnel. The pipeline can be instrumented, and efficiencies identified.



A structured pipeline is characterized by sign offs, checks, choke points, and automation.

The structured pipeline is oriented towards a particular shot, scene, or asset type.

The pipeline can be managed by non technical personnel. The pipeline can be instrumented, and efficiencies identified.

This model has flaws. If too many processes converge in a single step, the entire pipeline's efficiency is limited by the ability of that step to integrate the incoming assets. If a particular part of the pipeline is blocked, the entire production can halt. Worse, iteration (or rework) at the wrong point in the pipeline can invalidate previously completed parallel streams further down the pipeline, and it can be difficult to re-coordinate the pipeline flow to get the results correctly delivered to integrative steps.

The structured pipeline does not support experimentation with an asset; for efficiency, it is necessary that design be completed up front, before the pipeline is set in motion.

Data formats should be well documented, and robust translations must exist in order to support the kind of automation tools that support the structured pipeline.

## Type 3: Modular

- Special case of the structured pipeline.
- Portions of the pipeline can be swapped out quickly, reordered, or re-purposed with little difficulty.

face tomorrow



a modular pipeline is a special case of the structured pipeline.

The modular pipeline is strictly feed forward.

Like the ad hoc pipeline, the modular pipeline supports the entire production at the same time

An important aspect of the modular pipeline is that since it feeds strictly forward, there is no possibility of a backup. If iteration or experimentation occurs, these work products are introduced into the production stream, and they move forward to the finish line. Iteration is therefore isolated to a “spur” on the assembly line, which allows all other portions of the pipeline to keep moving.

The modular pipeline functions most smoothly when data formats are uniform.

Portions of the pipeline can be swapped out quickly, reordered, or re-purposed with little difficulty.

It's important to distinguish stream based steps which can be piped, from file based steps, and it's furthermore important that stream based protocols are well defined so that pieces of the pipeline can be reordered.

## Demo Reel

- Example of a modular pipeline in action

face tomorrow   
SIGGRAPH 2007

## Supporting the Rendering Pipeline

- To fully support a modular pipeline, the rendering pipeline itself must evolve to accommodate it.

## State and Geometry Driven Renderers

- State management APIs abstract an idealized model of the hardware as a state machine.
- It exposes the management of that virtual hardware to the graphics programmer.
- The data that moves through this pipeline is state packets and geometry.

face tomorrow 

State management APIs abstract an idealized model of the hardware as a state machine.

It exposes the management of that virtual hardware to the graphics programmer.

The data that moves through this pipeline is state packets and geometry. The pipeline needs to have a facility for full scene analysis so that packets can be optimally ordered.

This type of pipeline and rendering system is focused more on raw performance, and less on iteration.

A technique that helps out here is the notion of a sandbox where the artist can supply individual assets in order to avoid needing to wait for a reprocessing of the full scene.

## Scene Management Based Renderers

- Scene management engines are sometimes called retained mode renderers.
- The pipeline for a scene based renderer is tightly coupled to the rendering library.

face tomorrow SIGGRAPH2007

Scene management engines are sometimes called retained mode renderers.

Several modern render engines fall into this category. These engines are the most intrusive into your application program or game.

In general, these systems are only extensible through introducing complexity. More nodes and managers must be introduced, and then the interactions and marshalling of existing nodes and managers must be accomplished through delegation, spoofing, or rewriting the core systems.

The pipeline for a scene based renderer is tightly coupled to the rendering library. By necessity, the data formats are tightly coupled to runtime structures, and may be brittle in the sense that a runtime change may force a file format change, which might force a reprocessing of all assets to accomodate. Versioning tools that can read old formats and migrate them forward help here.

## Shader Systems Based Renderers

- A scene is described as a light transport problem, as in Renderman.
- Aside from the creation of shaders, the most important aspect of the pipeline is the assignment of shaders to geometry.

face tomorrow  
SIGGRAPH 2007

A scene is described as a light transport problem, as formulated originally by Kajiya. The predominant example of an offline shader based render engine is Renderman. The DirectX FX system is an example of a realtime shader based renderer.

Aside from the creation of shaders, the most important aspect of the pipeline is the assignment of shaders to geometry.

The pipeline needs to get geometry out of the DCC tool as quickly as possible, with as generic information as possible because the bulk of the work will be on the development of the shaders themselves, the assignment of different shaders to portions of the geometry, and to overrides of particular attributes on particular instances.

In this kind of a system, care must be taken that changes to the source geometry do not invalidate shader assignments made farther down the pipe, or that these downstream dependencies can be caught and validated without too much effort by the artist.

## Submission Engines

- What to draw is presented anew to the render engine every frame, in the form of a list of things to render, and composite operations.
- Submission APIs wrap a state management API, but do not expose the state management API in any way to the programmer.

face tomorrow  
SIGGRAPH 2007

Submission based engines combine features of scene management systems, and shader systems. Every frame, what to draw is presented anew to the render engine, in the form of a list of things to render, and composite operations.

Submission APIs wrap a state management API, but do not expose the state management API in any way to the programmer.

A well developed submission engine will provide a compile step, like an OpenGL display list. These precompiled submission lists potentially concatenate state information in a manner that allows fastest possible submission to the state management API. For example, on Xbox or X360, the compilation step bakes out a monolithic state block for the push buffers.

What is fundamentally different from a shader system is that it is up to the submission engine to manage and bind shaders and hardware state as appropriate. What is fundamentally different from a scene management system is that the described scene is not managed via a persistent scene model visible to the programmer. In other words, there are no nodes to manage, and the scene is presented to the submission engine fresh every frame. The submission engine can cache, hint, and manage state internally to optimize the submissions.

Pipeline implications...

## Data Design

- Data design is a key element of a pipeline.  
The important characteristics of good data design are:
- Uniformity
- Independence
- Separability
- Tolerance
- Longevity

face tomorrow 

## Uniformity

- One set of quirks to support through the pipeline rather than accomodations for various parsers and back end stores in each pipeline component.
- Uniform data design implies that all data of a similar kind is stored in the same way.

face tomorrow



A uniform and consistent data design is necessary to support a modular style pipeline. The benefits of this are that a single library and set of data bindings are sufficient to support any tool or portion of the pipeline. Processing occurs through a known set of operations, and the supported set of operations can be quality controlled and unit tested. As a result, there is one set of quirks to support through the pipeline rather than accomodations for various parsers and back end stores in each pipeline component.

Uniform data design implies that all data of a similar kind is stored in the same way. For example, all data objects might contain a vector of attributes, a vector of contained data objects, and a vector of relationships the data object owns. Each attribute might be described as a name, a value, and a type.

## Independence

- The second tier of the data format describes the relationships of the data objects, for example through a schema, or a byte code stream.
- Factor data transformations properly

face tomorrow

SIGGRAPH2007

The second tier of the data format describes the relationships of the data objects, for example through a schema, or a byte code stream. This tier might specify that a transform data object contains a vector of three floats, named “translation”, and more. It might also specify that a renderable geometry data object contains a transform object, a material object, and a geometry object.

When data is transformed, dependent changes should be atomic

Don't lose data that might need to be recovered later. For example, it is a mistake to tristrip a mesh if a later stage in the pipeline requires knowledge of mesh connectivity, or access to the source quads in the original mesh.

On the other hand, if a processing stage modifies data fundamentally, this should occur early in the pipeline. An example is the case where vertices need to be duplicated to break a mesh down by material. This process modifies the index buffers, which has a large downstream impact.

## Tolerance

- Garbage In == Garbage Out
- Watch for separable but dependent data

face tomorrow

SIGGRAPH2007

If the data is carefully designed, any tool in the pipeline should be able to read the data in, process the portions it knows about, and write the data out again without breaking any of the unknown portions. An early simple example of such a system is the AIFF file format. AIFF did not go far enough however, as the contents of individual chunks were arbitrary.

The pipeline needs to account for separable but dependent data that has downstream impact. A typical examples is lightmaps which need to be regenerated after modifications to geometry or light placement.

## Longevity

- If tools in the pipeline can derive relevant data via known conventions they can be independent of data format changes provided required data still exists.

face tomorrow



If the second tier is well designed, tools in the pipeline can derive relevant data via known conventions, and be largely independent of data format changes provided required data still exists. One way this can be accomplished is via a naming convention (the translation attribute is always called “translation”), or semantic tagging (the translation component is tagged with TRANSLATION). Neither one of these is perfect. Someone might decide that “translation” is a boolean type indicating that an object’s text needs translation into another language. More than one attribute may be tagged TRANSLATION (for example, globalTransform and localTransform), yielding an ambiguous interpretation to a parser without further clues.

## The Importance of Look Development and Pre-production

- Look development keeps production costs down.
- Gear the technologies and development work in pre-production towards achieving the look.
- New development should not disrupt the pipeline.
- Don't make changes that mess with the developed look.

face tomorrow



Look development can help keep production costs down. During the pre-production phase, try to create prototype art using whatever packages and tools you can get your hands on. Try to push a few sample assets and scenes to final quality. Once this benchmark is established, gear the technologies and development work in pre-production towards achieving the look. Once baseline functionality is established, predicate new work on the whether it supports the look. At this point, save exploration for the next project. Given the sheer quantity and sophistication of assets required for products in this generation, development effort should be geared towards streamlining art production, not towards changing the art that is being produced.

In the case where a new development would cause data or process changes, make sure that the changes do not disrupt the pipeline. Make sure that the data changes are transparently incorporated into the pipeline, and that a batch reprocessing can occur without team downtime.

Above all, once production has started, don't make changes that mess with the developed look. Although you may have invented the coolest new lighting technique of all time, you risk incurring massive amounts of rework to other parts of the pipeline. In the worst case, an innovation may force shaders to be rewritten across the board, and all existing lighting setups to be invalidated. In the very worst case, the original look might be invalidated, sending the art team back to the drawing board. Save it for the next project!

# Principals of pipeline design

- Keep the pipeline unidirectional
- The pipeline should have an automated batch mode whereby an entire asset type and all downstream dependencies can be regenerated.
- It's therefore important that each processing stage can run headless. Each small tool should be able to run without a GUI.
- Command line interfaces should be rich enough to support what you want to do. Besides exposing all relevant parameters, or relying on a response file, it should be possible to set up independent inputs and outputs, in other words, it should be possible that an output file not overwrite the input file.
- The pipeline should have traps for bad data, incomplete processing, and error conditions occurring during processing.



## DirectX in the pipeline

- Acceleration of computation
- Approximation of final render
- Fidelity of assets
- Demo Reel

face tomorrow  SIGGRAPH 2007

## The Future

- Sophisticated simulation
- Deep compositing
- Participating Media
- Complex character rigs and control
- Tessellation

face tomorrow  SIGGRAPH 2007





*Debugging Direct3D 10 Applications* (Chuck Walbourn): The new Direct3D 10 technology provides a series of new debugging and profiling mechanisms for graphics code, and leverages tools like *PIX for Windows* to provide shader-debugging functionality. This topic covers the use of these facilities and demonstrates how to use the new tools for debugging Direct3D 10 applications. [30 minutes]

## Debugging and Symbols

- Finding crashes with a debug build is often easy
- Finding real-world crashes is not
- Symbols are the key
  - On 32-bit applications, they are critical to clean ‘stacks’ for release optimized code
  - Mini-dumps (.dmp files) are amazingly useful if you have adequate symbols

face tomorrow  SIGGRAPH 2007

## Symbol Server

- Make sure you are generating debug information even in 'release' configurations
- Keep them for each published version
- For everything else, Microsoft Symbol Server is the place to start
  - Visual Studio 2005 already configured to use it
  - Otherwise follow instructions at

<http://www.microsoft.com/whdc/devtools/debugging/debugstart.mspx>



## Video Drivers

- With XPDM, a crash in the driver typically brought down the whole system: BSOD!
- WDDM breaks the driver into a User Mode and a Kernel Mode portion
  - Crashes in the User Mode portion just terminate your application, which you can then restart or debug
  - Even kernel-mode crashes can sometimes be fixed with an automatic reset
  - User-mode driver crashes will show up inside your process when debugging, which might give you a hint as to what's triggering it



## Direct3D 9 Debugging Support

- DirectX SDK installs a ‘Developer Runtime’
- Developers can select “Debug” vs “Release” versions of Direct3D system-wide
  - DirectX Utilities -> DirectX Control Panel
  - Debug versions generates various levels of output to `OutputDebugString` captured by debugger
  - Developers that leave “Debug” versions active run all Direct3D applications slowly
- Link to debug version of D3DX9 library for debug output

face tomorrow



## Direct3D 10 Debugging Support

- DirectX SDK installs ‘SDKDebugLayers’
- Developer can enable and control this ‘debug layer’
  - Programmatically via the application
  - Per-application use via the DirectX Control Panel

## SDK Debug Layer

- Direct3D 10 Runtime minimizes validations to improve performance
- Debug Layer will perform extensive validations to uncover usage, performance, and stability problems
  - Corruption and Errors are the most serious
  - Warnings generally should be fixed
- Other tools, such as *PIX for Windows*, often assume you are already 'debug layer clean'
  - i.e. would not generate Corruption or Errors

Face tomorrow



# Enabling Debug Layer

```
DWORD dwFlags = 0;

#ifndef DEBUG
dwFlags |= D3D10_CREATE_DEVICE_DEBUG;
#endif

HRESULT res = D3D10CreateDeviceAndSwapChain( NULL,
    D3D10_DRIVER_TYPE_HARDWARE, NULL, dwFlags, D3D10_SDK_VERSION,
    &sd, &g_pSwapChain, &g_pDevice );

if ( FAILED(res) ) // Error Handling
// Would fail with #define DEBUG if run on a system without
// the DirectX SDK installed to get the SDKDebugLayer
```

SIGGRAPH2007

## Controlling the Debug Layer

- Just creating with `D3D10_CREATE_DEVICE_DEBUG` will turn on most messages
- You can get more control by obtaining the debug interfaces from your `ID3D10Device` instance
  - Uses standard COM QueryInterface techniques
  - You will fail to get the debug interfaces if the device was not created with the above flag

## ID3D10Debug

- This interface can enable and control some special debug features
- These features impact specific operations: draws, clears, copy/update resource, generate mips, and MSAA resolves
  - D3D10\_DEBUG\_FEATURE\_FINISH\_PER\_RENDER\_OP
    - Causes a wait until the GPU finishes the command
  - D3D10\_DEBUG\_FEATURE\_FLUSH\_PER\_RENDER\_OP
    - Causes these operations to call ID3D10Device::Flush
  - D3D10\_DEBUG\_FEATURE\_PRESENT\_PER\_RENDER\_OP
    - Causes a Present after each operation

## ID3D10Debug Example

```
ID3D10Debug *g_pDebug = 0;  
  
HRESULT res = g_pDevice->QueryInterface( __uuid( ID3D10Debug ),  
    reinterpret_cast<void**>( &g_pDebug ) );  
  
if ( FAILED(res) )  
    // Error Handling, could be because  
    // device was not created with "DEBUG"  
  
...  
  
if ( g_pDebug )  
    g_pDebug->SetFeatureMask(  
        D3D10_DEBUG_FEATURE_FINISH_PER_RENDER_OP );
```

SIGGRAPH2007

## ID3D10InfoQueue

- All Direct3D 10 debug output is sent via a centralized message queue
- Developers can get an interface to this queue and use it to
  - Silence categories of messages or specific messages
  - Pull out debug messages from the runtime
    - Could redirect them to application's debugging facilities
  - Add custom messages

## ID3D10InfoQueue Example

```
ID3D10InfoQueue *g_pDebugInfo = 0;

HRESULT res = g_pDevice->QueryInterface( __uuid(
    ID3D10InfoQueue ), reinterpret_cast<void**>( &
    g_pDebugInfo ) );

if ( FAILED(res) )
{
    // Error Handling, could be because
    // device was not created with "DEBUG"
}

...
```

SIGGRAPH2007

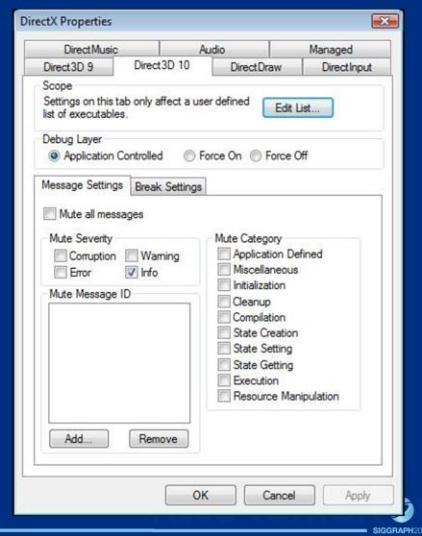
## ID3D10InfoQueue Example

```
...
// Silence a specific message
if (g_pDebugInfo) {
    D3D10_INFO_QUEUE_FILTER filter;
    memset(&filter, 0, sizeof(filter));
    D3D10_MESSAGE_ID i[] = { D3D10_MESSAGE_ID_<xxx> };
    filter.DenyList.NumIDs = 1;
    filter.DenyList.pIDList = i;
    g_pDebugInfo->PushStorageFilter( &filter );
}
```

SIGGRAPH2007

# DirectX Control Panel

- Instead of having to change the code, the control panel can force the “DEBUG” creation flag and/or mute messages
- Only applies to specific applications in specific locations set with “Edit List...”
- Requires administrator rights to use this utility!



## Optional Direct3D 10 Features

- A common source of ‘bugs’ on Direct3D 9 was using a capability without checking the current device caps
- For the most part, this is completely eliminated in D3D10
  - Only a few formats usages are optional
  - Multi-Sample Anti-Aliasing (MSAA) of >1 sample is optional
  - Check with `ID3D10Device::CheckFormatSupport`
  - Debug layer should complain if you are using an unsupported format for the current device
  - Still want to make sure to try the application on multiple vendor’s cards, especially for performance evaluation

face tomorrow



## Reference Device

- As with previous versions of Direct 3D, the DirectX SDK includes a ‘reference device’
  - Full software implementation of a Direct3D 10 device
  - Not written for performance, but only for compliance
    - (i.e., for anything but a postage stamp it can be painfully slow)
  - Can be useful in determining if a bug is a driver problem or something else
- Can either create it manually or force it with tools like *PIX for Windows*

# Reference Device Creation

```
DWORD dwFlags = 0;  
#ifdef DEBUG  
dwFlags |= D3D10_CREATE_DEVICE_DEBUG;  
#endif  
  
D3D10_DRIVER_TYPE drvType = D3D10_DRIVER_TYPE_HARDWARE;  
#ifdef USEREF  
drvType = D3D10_DRIVER_TYPE_REFERENCE;  
#endif  
  
HRESULT res = D3D10CreateDeviceAndSwapChain( NULL, drvType, NULL, dwFlags,  
    D3D10_SDK_VERSION, &sd, &g_pSwapChain, &g_pDevice );  
if ( FAILED(res) ) // Error Handling  
// Both DEBUG and USEREF can fail if the DirectX SDK is not installed on this  
machine
```

SIGGRAPH2007

## FXC HLSL Compiler

- Can be useful to look at the resulting assembly for a shader to find bugs or performance problems
- Easiest way to do this is to use the **FXC .EXE** command-line tool to compile shader files

## PIX for Windows

- DirectX SDK Utility for performance analysis and debugging
- Frame Capture data can be used to examine and debug Vertex, Pixel, and Geometry shaders
  - Can examine shader assembly and HLSL source

face tomorrow



SIGGRAPH 2007

# PIX for Windows

DEMO

face tomorrow 

## Summary

- Use the SDK Debug Layer routinely to quickly find problems with API usage and other issues
- When faced with a difficult bug, consider using the SDK debug layer interfaces
- Test on multiple vendor's hardware
- Use *PIX for Windows* shader debugging



# Direct3D 10 Performance Optimization

Chas. Boyd

Microsoft Corp

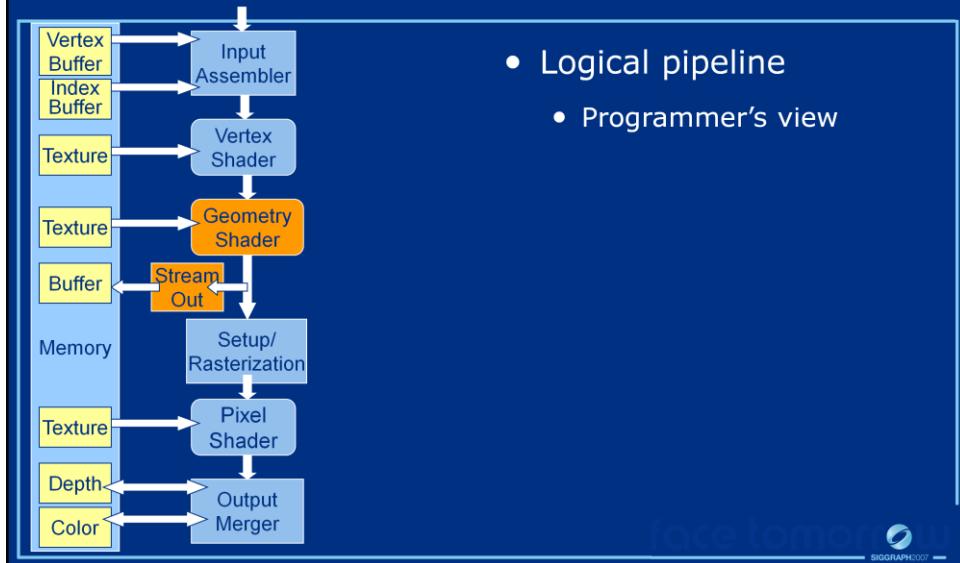
*Performance Tuning for Direct3D 10* (Chas Boyd): The new Direct3D 10 architecture is designed to enable applications to minimize the CPU overhead of rendering. This session reviews design strategies and best-practices to maximize performance on the new API and hardware. This topic reviews the standard usage idioms expected for Direct3D 10, with comparisons to existing Direct3D 9 code patterns. [60 minutes]

## Session Outline

- Drilling Down on 10
  - Feeding the pipeline
  - Managing pipeline state
  - Linking it all together
  - Presenting the result – DXGI
  - General Performance Tuning

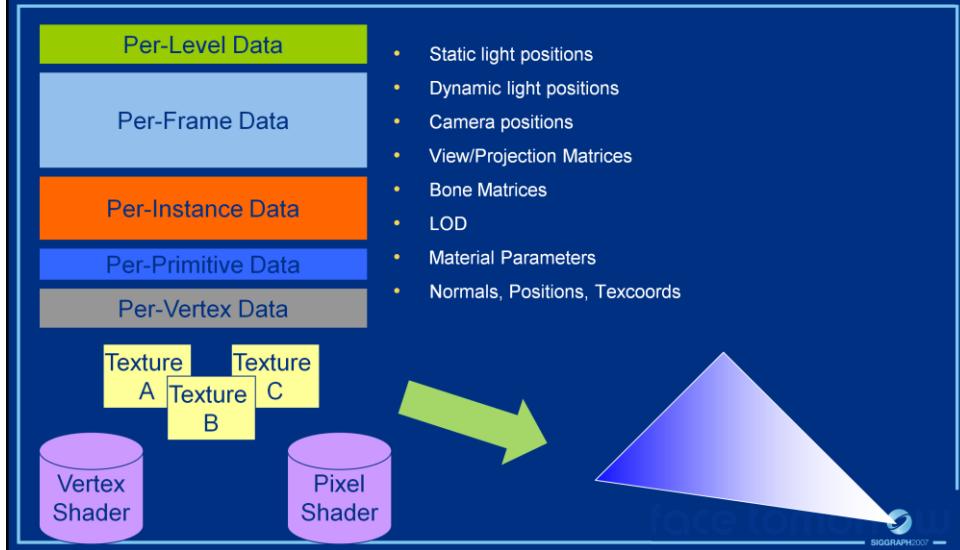


# Direct3D 10 Pipeline



Now I'll quickly walk through the pipeline stages

# Shading A Triangle



Now I'd like to take a different turn and describe how a large application, such as a game, draws a scene

In total there can be quite a large amount of state associated with drawing a single triangle

[click] There is static data for the scene such as fixed light positions

[click] There is also scene state that changes each frame such as dynamic light and camera positions

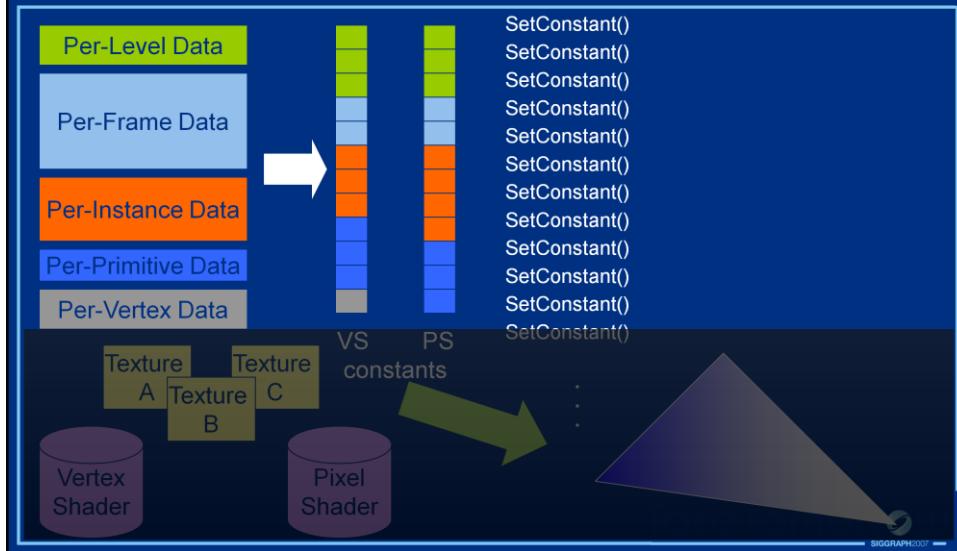
[click] Then there is the per-object instance data, for example if it is a moving character, it might include bone matrices or level of detail data

[click] Then there is the per-primitive data such as surface material properties

[click] Finally the per-vertex data such as normals , postions, etc

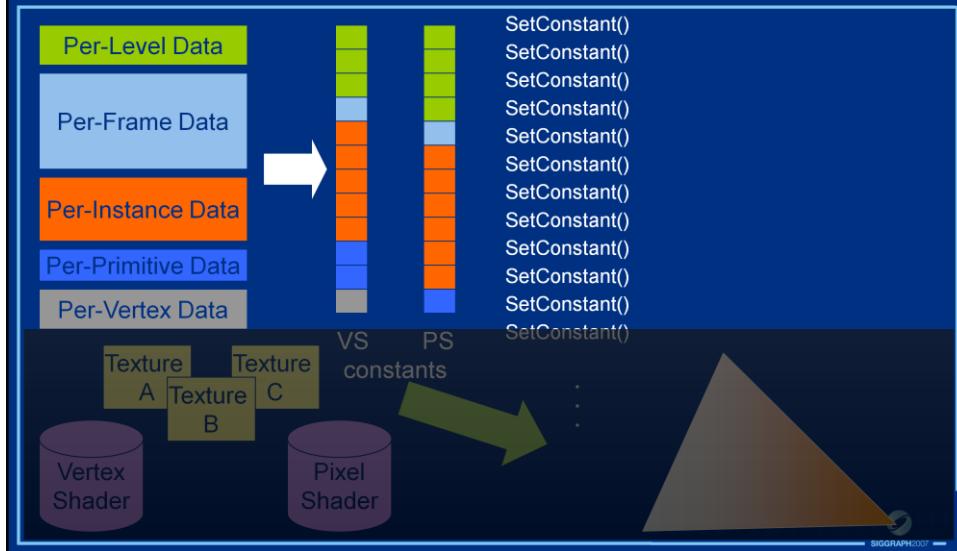
Besides that, there are the texture maps, and vertex, pixel, and now geometry shaders associated with the object.

## Constants in Direct3D 9



Now lets concentrate on this top set of state. Since the pipeline is now programmable. This state is stored in constant memory associated with each programmable stage. Each piece of state: matrix, vector, color, etc is set with an SetConstant API call to move it from the CPU to the GPU.

## Constants in Direct3D 9



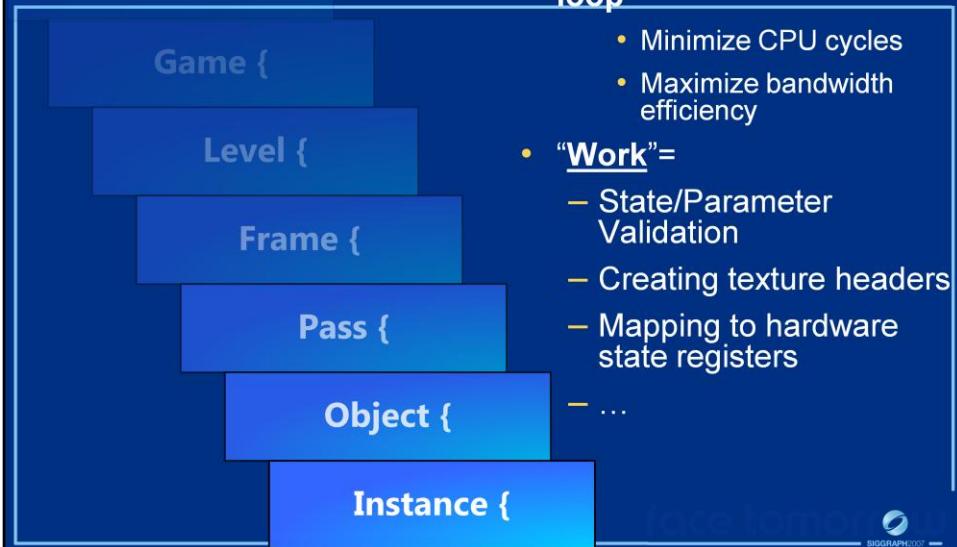
While it would be nice to keep that constant memory nicely arranged, so we don't have to update all of the constant memory all of the time

The amount of state data for each object can vary greatly so it makes it difficult to keep pieces of state in fixed positions

And the result is that the constant memory is rewritten all of the time.

# Frontload the Work

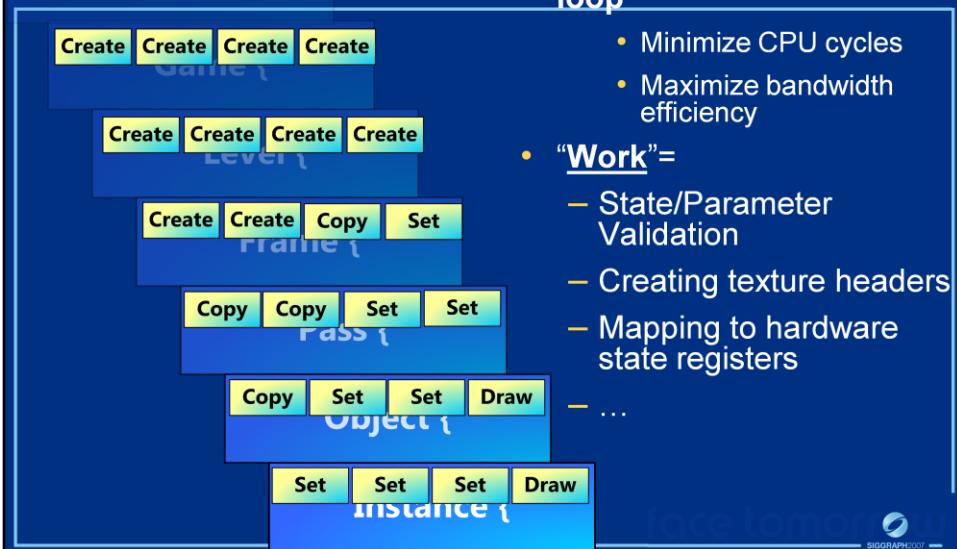
Content Creation {



SIGGRAPH 2007

# Frontload the Work

Content Creation {



SIGGRAPH2007

## Frontload the work

- Move work into the **outer loop**
  - Minimize CPU cycles
  - Maximize bandwidth efficiency
- “Work” =
  - State/Parameter Validation
  - Creating texture headers
  - Mapping to hardware state registers

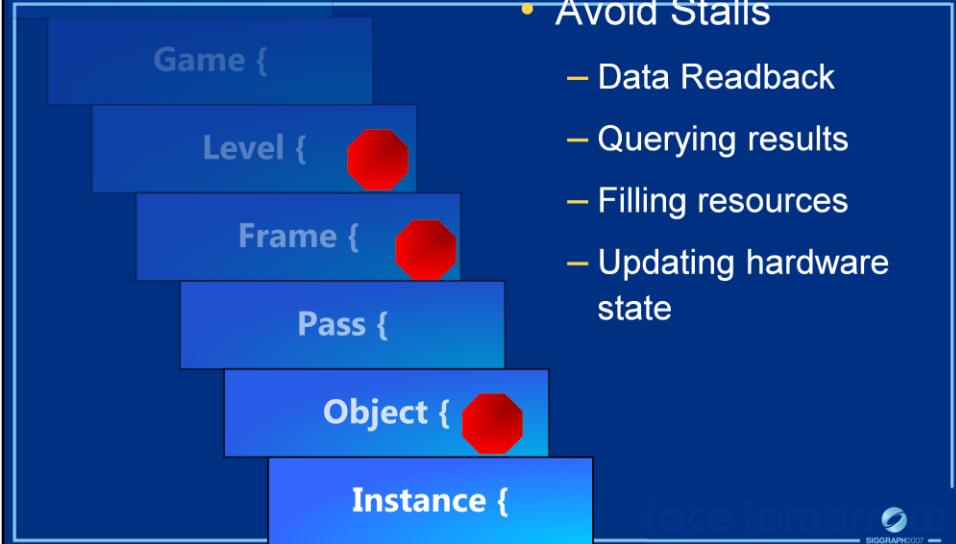


Frontload the work



# Pipeline Everything

Content Creation {

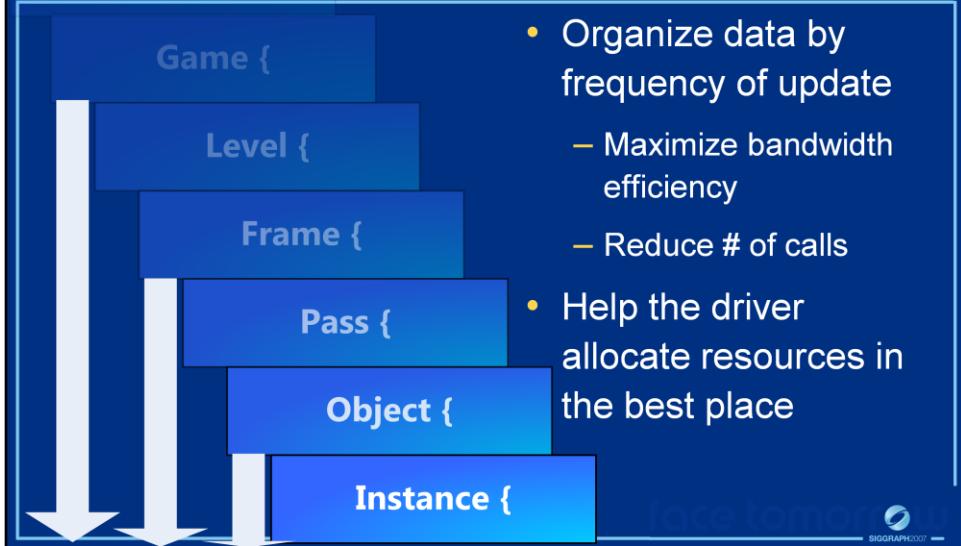


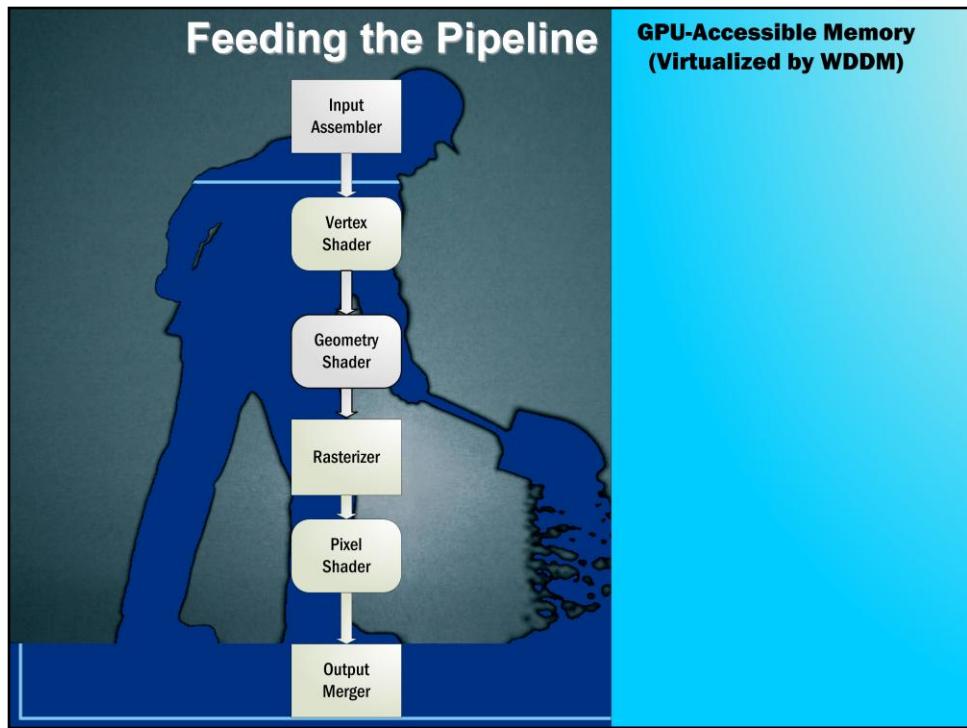
face tomorrow

SIGGRAPH2007

# Optimize for Frequency of Update

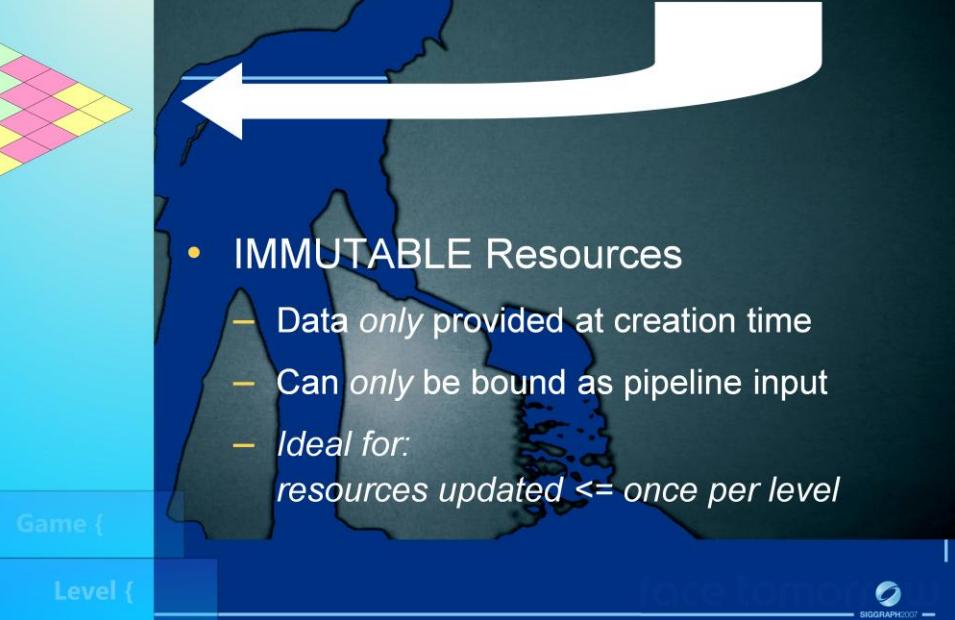
Content Creation {





## Feeding the Pipeline

ID3D10Device::CreateTexture2D(pDesc, *pInitialData*)



## Feeding the Pipeline

ID3D10Resource::UpdateSubresource(...*pSRCData* , ...)

- **DEFAULT Resources**
  - Data provided at creation time and/or via UpdateSubresource
  - Can be used as pipeline output
    - Can also be updated via Copy...()
  - Ideal for:  
*resources updated < once per frame*



## Feeding the Pipeline

Map(D3D10\_MAP\_WRITE)

- DYNAMIC Resources
  - Updated via Map()/Unmap()
  - Can't be used as pipeline output
  - Can be updated via Copy...()
  - *Ideal for:*  
*resources updated > once per frame*

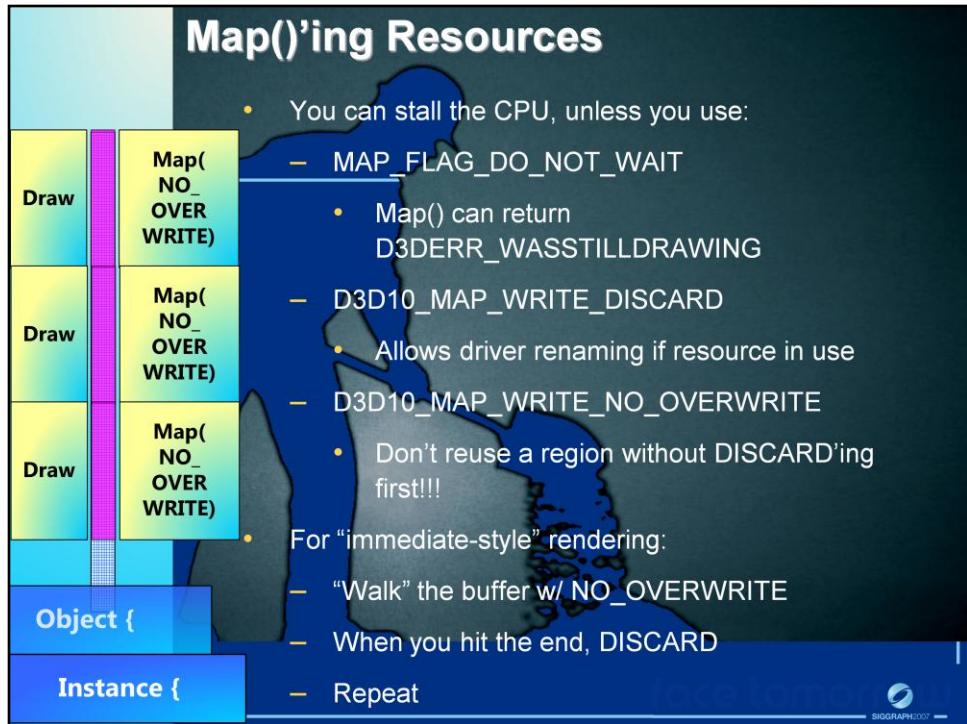
Pass {

Object {

SIGGRAPH2007

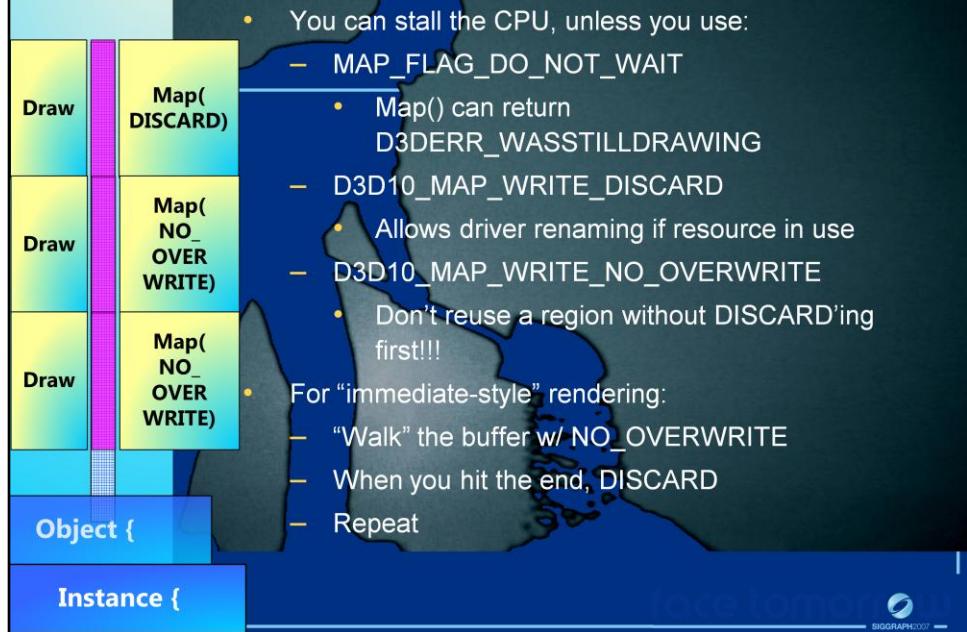
## Map()'ing Resources

- You can stall the CPU, unless you use:
  - MAP\_FLAG\_DO\_NOT\_WAIT
    - Map() can return D3DERR\_WASSTILLDRAWING
  - D3D10\_MAP\_WRITE\_DISCARD
    - Allows driver renaming if resource in use
  - D3D10\_MAP\_WRITE\_NO\_OVERWRITE
    - Don't reuse a region without DISCARD'ing first!!!
- For "immediate-style" rendering:
  - "Walk" the buffer w/ NO\_OVERWRITE
  - When you hit the end, DISCARD
  - Repeat

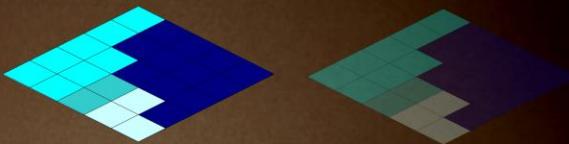


SIGGRAPH2007

## Map()'ing Resources



## Harvesting Pipeline Results

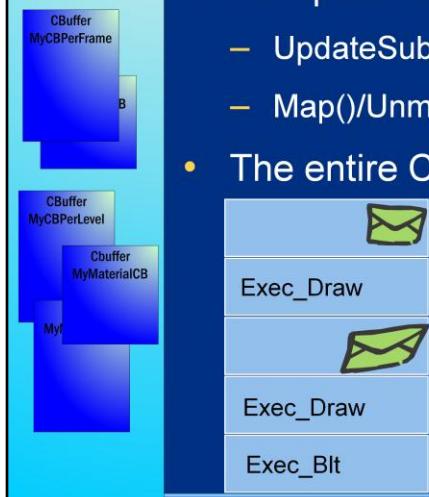


- STAGING - The readback fast-path
  - Copy() GPU result to a staging resource
  - Map() the staging resource for read
  - DO\_NOT\_WAIT or you will stall the CPU

SIGGRAPH2007

# Constant Buffer Update

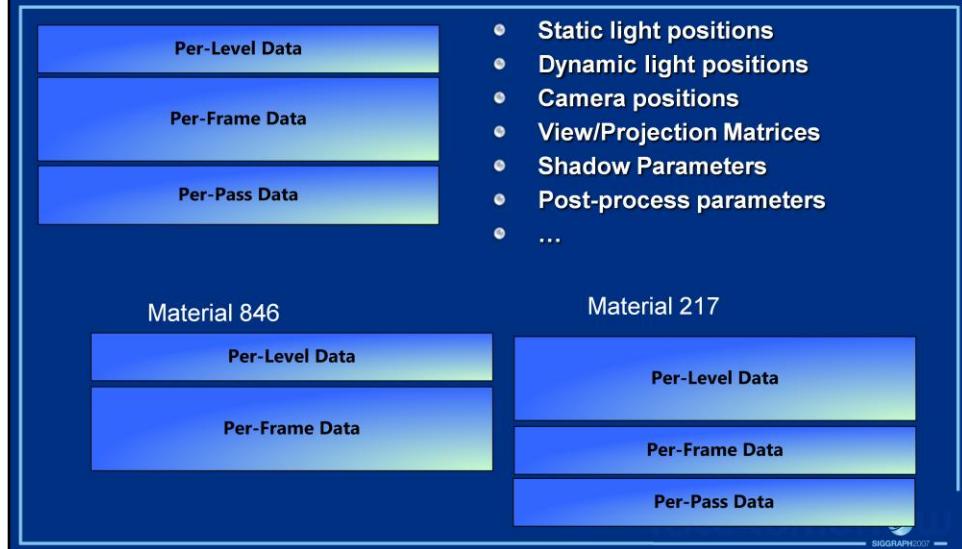
- CB update uses high-bandwidth paths
  - UpdateSubresource
  - Map()/Unmap() with Discard
- The entire CB must be updated



- Updates of small CBs can still be inlined into the command buffer

SIGGRAPH 2007

# Analyzing your Constants

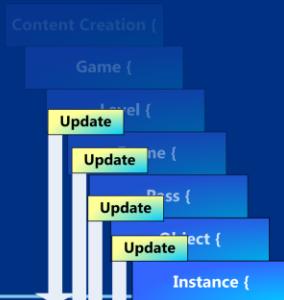


# Optimizing Constant Organization

- 1<sup>st</sup>: Organize by frequency of CPU update
  - Minimize API calls and CPU→GPU bandwidth

Cbuffer MyCBPerLevel

Cbuffer MyCBPerFrame



face tomorrow



# Optimizing Constant Organization

- 1<sup>st</sup>: Organize by frequency of CPU update
  - Minimize API calls and CPU→GPU bandwidth

Cbuffer MyCBPerLevel

Cbuffer MyCBPerFrame

- 2<sup>nd</sup>: Organize by shader usage as *needed*
  - Maximize independence of materials
  - Squeeze into the 4096 float4 buffer ☺

Shader A

Shader B



SIGGRAPH2007

# HLSL Constant Packing

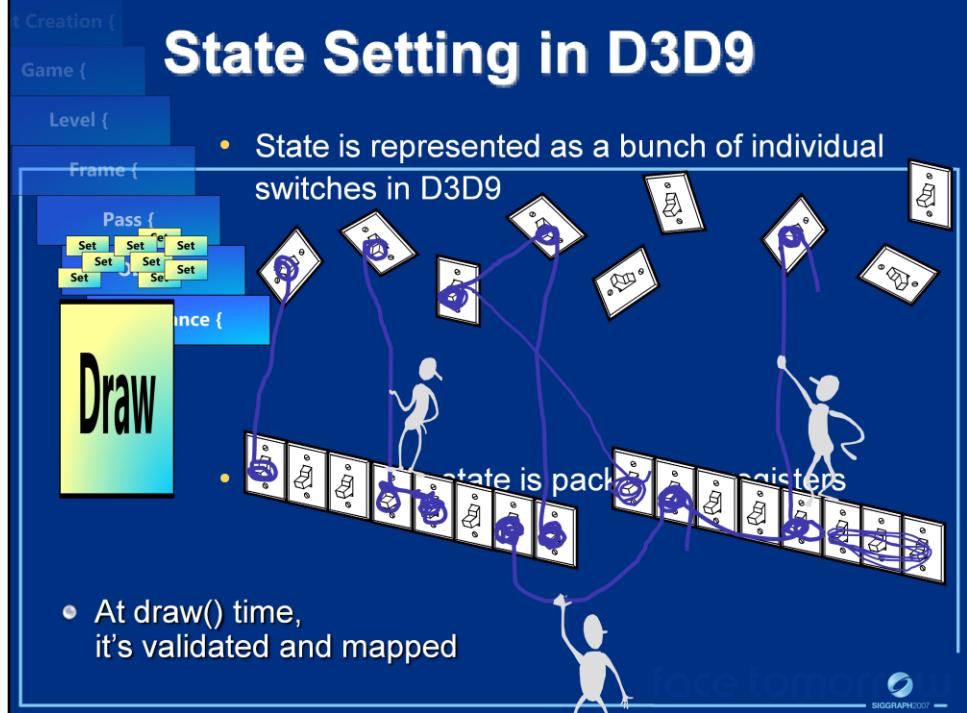


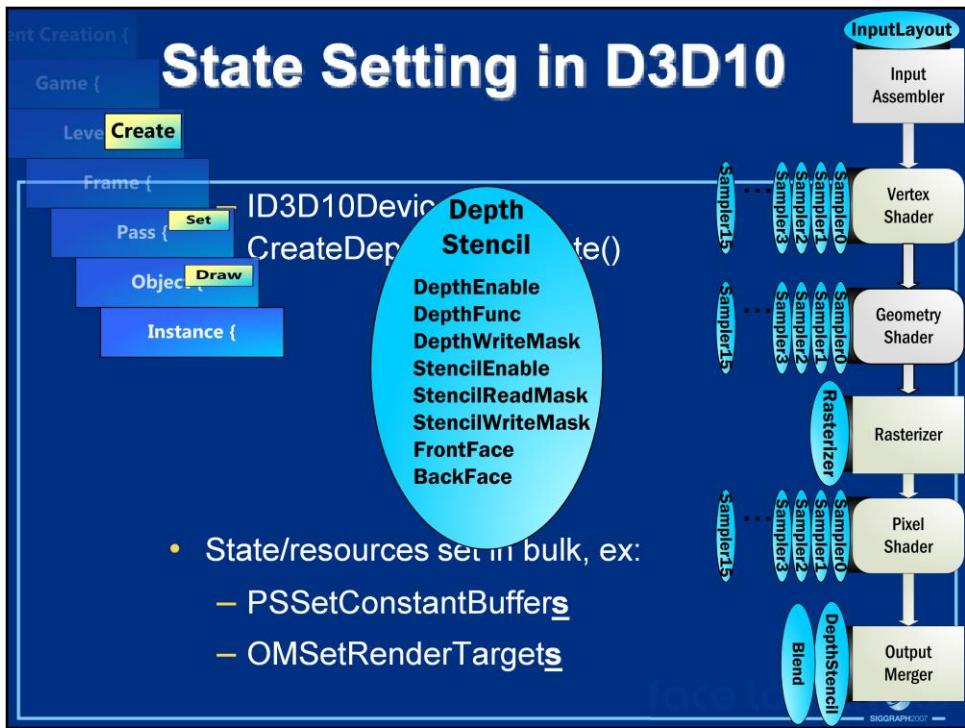
- Applies to tbuffers and cbuffers
- Variables will be packed together into vec4's, but will not cross vec4 boundaries
- Structs & array elements always start a new vec4
  - Cast for more aggressive array packing
    - Less memory bandwidth,
    - more ALU's for addressing:

```
float4 arr[16];  
static float2 unpackedarr[32] = (float2[32])arr;
```

- Pack yourself via : packoffset() keyword
- Use Effects or ID3D10ShaderReflection to reflect this layout

SIGGRAPH 2007





## D3D10: Ref Counting

- Unlike D3D9, Set()'s don't change ref count
- When ref count drops to zero, the object will be *unbound from the pipeline and destroyed*
  - → App must hold a reference
    - Get()'s increase ref count
  - Debug layer will warn you
- Consider this when porting from 9

```
pDevice->CreateRasterizerState( ... , &pRasterizerState );
pDevice->RSSetState( pRasterizerState );
pDevice->RSGetState( &pCurRasterizerState );
// pCurRasterizerState will be equal to pRasterizerState.

pCurRasterizerState->Release();
pRasterizerState->Release();

// App released the final ref on this object → it's deleted.
pDevice->RSGetState( &pCurRasterizerState ); // (NULL)
```



0  
+1  
+1  
+1  
-1  
-1

SIGGRAPH 2007

## D3D10 Runtime Filtering

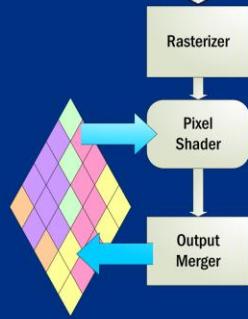
- The runtime filters duplicate state objects and input layouts on Create()
  - Many API objects → 1 DDI Object
- The runtime filters redundant scalar set()'s
  - i.e. SetDepthStencilState
  - Not SetScissorRects()
- Get()'s cost 1 addref/release
- **Takeaway:**
  - Tie state objects to your material abstraction
  - For best batch performance, avoid redundant filtering



## D3D10 Hazard Detection



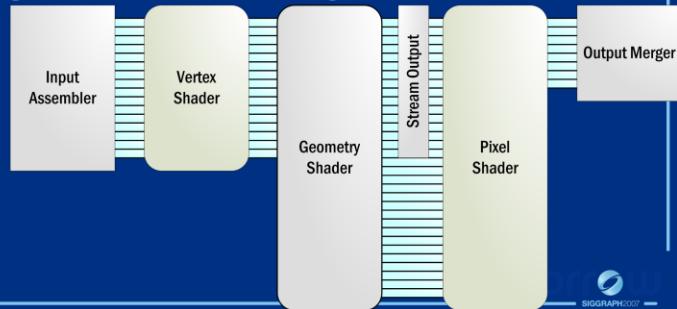
- Set()'s, Draw()'s, Copy()'s return void
  - No error checks in your code
- The same *subresource* can't be simultaneously bound as pipeline input and output
- The runtime will automatically unbind the resource elsewhere
  - Debug layer can inform you
- Unset aggressively to avoid stall



SIGGRAPH 2007

## Fast Interstage Linkage

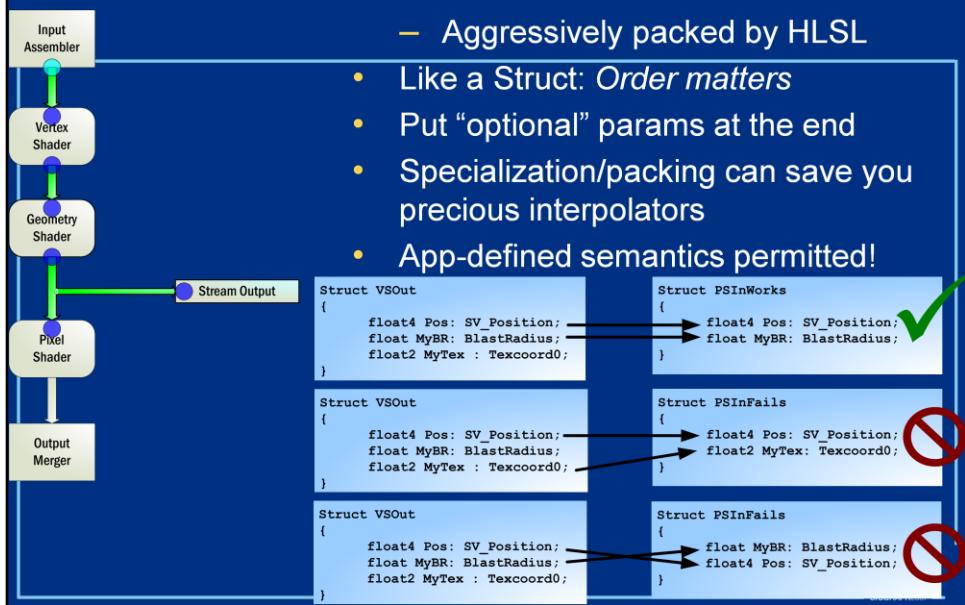
- Direct3D10 API Design Imperative:  
No draw-time fix-up required
- Fixed generic register banks connect stages
- Shader linkage enforce via ‘Signatures’



SIGGRAPH2007

# Signatures

- Correspond to shader parameter declarations
  - Aggressively packed by HLSL
- Like a Struct: *Order matters*
- Put “optional” params at the end
- Specialization/packing can save you precious interpolators
- App-defined semantics permitted!



# Optimizing Shader Linkage

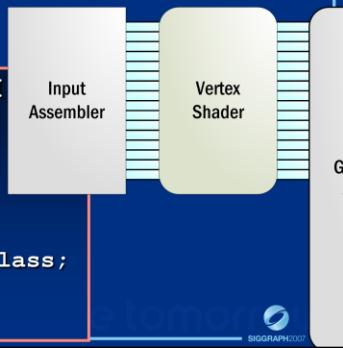
- Why Reduce Interpolators?
  - Fit in the hard limit (16 out of VS, 32 out of GS)
  - Saves critical on-chip bandwidth → More threads!
- Techniques
  - Barycentric Coordinates
    - Interpolants will be cached, but introduces latency
  - Specialize VS/GS/PS combos
    - #ifdef shader outputs
    - Use shader input mask to determine what's needed upstream (ID3D10ShaderReflection/ID3D10Effect)
    - Your only option in DX9, unless you simplify your shader



# Input Layouts

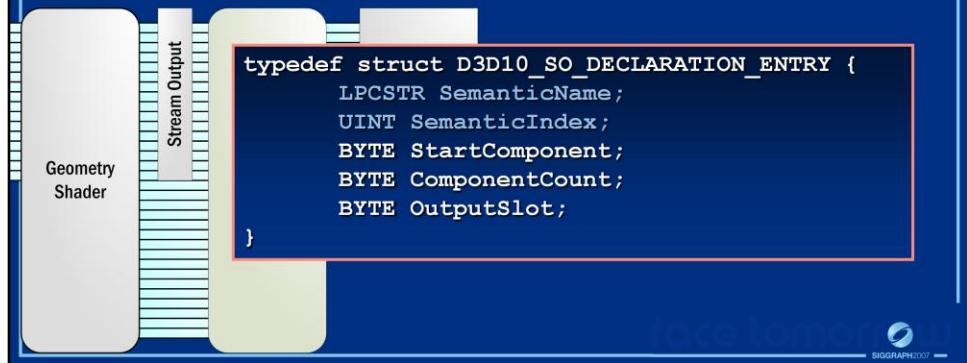
```
CreateInputLayout(
    const D3D10_INPUT_ELEMENT_DESC * pInputElementDescs,
    UINT NumElements,
    const void * pShaderBytecodeWithInputSignature,
    ID3D10InputLayout ** ppInputLayout)
```

```
typedef struct D3D10_INPUT_ELEMENT_DESC {
    LPCSTR SemanticName;
    UINT SemanticIndex;
    DXGI_FORMAT Format;
    UINT InputSlot;
    UINT AlignedByteOffset;
    D3D10_INPUT_CLASSIFICATION InputSlotClass;
    UINT InstanceDataStepRate;
}
```



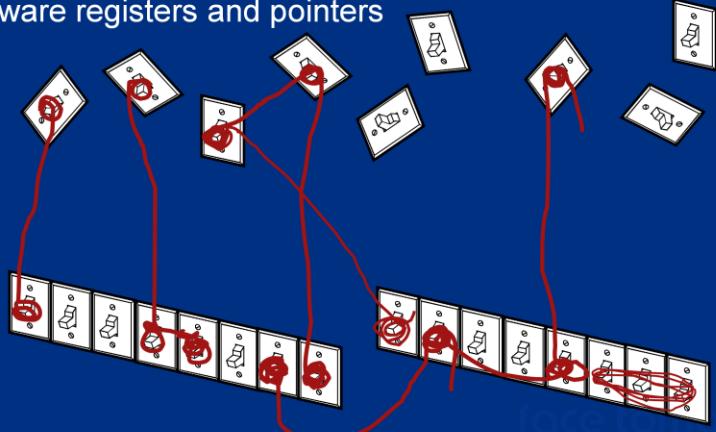
# Stream Output Layout

```
CreateGeometryShaderWithStreamOutput(  
    const void * pShaderBytecode,  
    const D3D10_SO_DECLARATION_ENTRY * pSODeclaration,  
    UINT NumEntries,  
    UINT OutputStreamStride,  
    ID3D10GeometryShader ** ppGeometryShader  
)
```



## What is a Material To D3D9 Hardware (State-Driven)

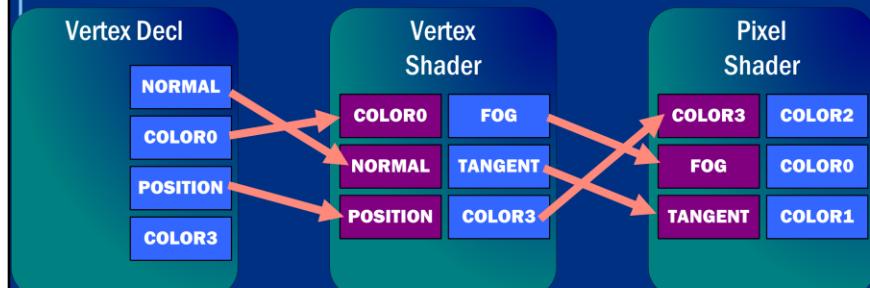
At Draw-time, tons of independent API render state and bindings must be validated in the runtime and mapped in the driver to hardware registers and pointers



SIGGRAPH2007

## What is a Material To D3D9 Hardware (State-Driven)

- Interstage linkage must be dynamically reconfigured

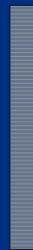


face tomorrow SIGGRAPH 2007

## What is a Material To D3D9 Hardware (State-Driven)

- Large #'s of constants must be sent down the command stream to reset a fixed set of hardware registers
- API calls required for almost all material transitions

SetConstant()  
SetConstant()  
SetConstant()  
SetConstant()  
SetConstant()  
SetConstant()  
SetConstant()



Single, shared  
constant  
register bank

SetStreamSource()  
SetStreamSource()

SetTexture() SetTexture()  
SetTexture() SetTexture()  
SetTexture() SetTexture()  
SetTexture() SetTexture()  
SetTexture() SetTexture()  
SetTexture() SetTexture()  
SetTexture() SetTexture()

## What is a Material to D3D10 Hardware (Data Driven)

- Constant Buffers filled as needed via high-bandwidth paths
- Pre-computed state objects and views stored in memory
  - handles sent down in command stream
- Parameterizations can be stored in memory, read in shader
  - Indexable, per-primitive/object/instance material info in buffers/CB's



face tomorrow



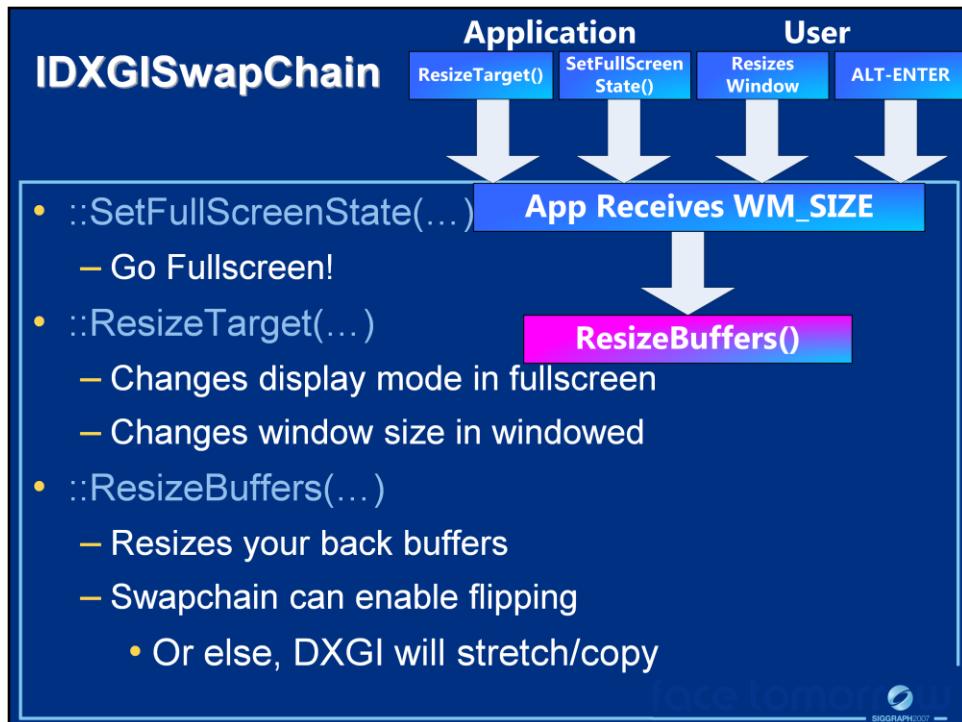
SIGGRAPH 2007

## DXGI – DirectX Graphics Infrastructure

- Factor shared, “2D” concepts out of the 3D render API
  - Adapters, presentation, shared surfaces, ...
- DXGI’s objects match hardware
  - **IDXGIAdapter** = the card:  
GPU, memory, DACs, ...
  - Adapters own **IDXGIOOutputs** = monitors
    - Mode list, Gamma Control,  
Presentation Statistics, ...
- Direct3D10 takes an adapter at startup
  - Naturally “headless” w/o a swap chain
- D3D10CreateDeviceAndSwapChain

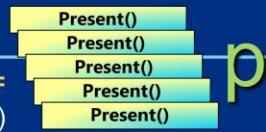


SIGGRAPH2007



# Measuring Performance

# of Present calls ( $p$ ) =  
(IDXGISwapChain::GetPresentCount())



# of frames presented to the display ( $f$ ) =  
(DXGI\_FRAME\_STATISTICS.PresentCount)



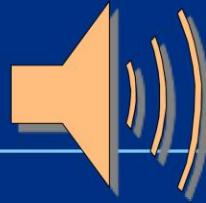
# of Vsync's ( $v$ ) =  
(DXGI\_FRAME\_STATISTICS.PresentRefreshCount)

- How far behind is the GPU?
  - # of frames in-flight =  $p - f$
- Have I glitched?
  - Yes, if  $\Delta f \neq \Delta v$
- Only relevant over a wide sampling



SIGGRAPH 2007

## Sync'ing to Audio



- DXGI\_FRAME\_STATISTICS.SyncQPCTime
  - The approximate value of QueryPerformanceCounter at the last vsync
- Combine with frame latency to skip ahead



## Performance Bottlenecks

- Draw-Call or Batch Overhead Limits
- Input Limits
  - Number of texels or vertices loaded per clock
- Output Limits
  - Number of pixels or vertices emitted per clock
- Compute Limits
  - Long shaders, few texture loads/samples
  - Triangle setup rate
  - Register pressure: number of temp registers used 

Most implementations have fundamental limits that can be attained. 32texels/clock or 24 pixels/clock

If you suspect a limit has been hit, change a parameter that would make it worse and see if performance drops.

If so, that may have been it, if no change (or little), try a different parameter.

Once a bottleneck is identified, a specific change can be made to address it.

## Register Pressure

- There are finite resources in the GPU:
  - Temporary registers used by running threads
  - State stored for suspended threads
  - Interpolators
- Too many temporary registers means fewer threads can run on the fixed register set
- It can also mean a reduced number of threads can be suspended
  - Which means less ability to hide latency of i/o



DX10 shader model 4 max register usage per shader is 1024 4-float temporaries.

No shader has ever used this many, and probably would not get any parallel processing if it did.

You can find out the number of temporary registers a shader uses by looking at the disassembly.

(in some cases the driver may be able to reduce this slightly)

The compiler optimizes for register re-use as much as possible, but you can always help it out.

Shader model 4 should have no issues with 4 simultaneous temporaries, earlier models (on some parts) may run better with 3 or less.

10s of threads can be running simultaneously, and 100s could be suspended.

## Fixes for Performance

- Compute Bound:
  - Reduce register usage (check .asm) via simpler algorithm or more table-lookup methods
  - Reduce use of complex instructions
- Draw Call Limited
  - Use more instancing
  - Use shader to select between states inside a single draw call
    - Via flow control, array inputs, etc.

## Summary

- Manage constants efficiently by grouping based on frequency of use
- Don't spend lots of time filtering out redundant state changes
  - Ideally just never generate them...
- Measure performance frequently
  - Don't optimize until it is a proven problem

face tomorrow  SIGGRAPH 2007

This document is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

# API Features (Direct3D 10)

The Direct3D 10 graphics pipeline represents a fundamental architecture change, rebuilt from the ground-up in hardware and software to power the next-generation of games and 3D multimedia applications. It is built upon the Windows Vista Display Driver Model (WDDM) infrastructure, enabling new performance and behavioral enhancements and guaranteeing of full virtualization of GPU memory.

Developers familiar with Direct3D 9 will discover a series of functional enhancements and performance improvements in Direct3D 10, including:

- The ability to process entire primitives (with adjacency), amplify and de-amplify data in the new geometry shader stage.
- The ability to output pipeline-generated vertex data to memory using the stream output stage.
- New objects and paradigms provided to minimize CPU overhead spent on validation and processing in the runtime and driver.
  - Organization of pipeline state into 5 immutable state objects, enabling fast configuration of the pipeline.
  - Organization of shader constant variables into constant buffers, minimizing bandwidth overhead for supplying shader constant data.
  - The ability to perform per-primitive material swapping and setup using a geometry shader.
- New resource types (including shader-indexable arrays of textures) and resource formats.
- Increased generalization of resources in memory and ubiquity of resource access - resource views enable interpretation of resources in memory as different types or representations.
- A full set of required functionality: legacy hardware capability bits (caps) have been removed in favor of a rich set of guaranteed functionality. To enable this and other design improvements, the Direct3D 10 API only targets Direct3D 10-class hardware and later.
- Layered Runtime - The Direct3D 10 API is constructed with layers, starting with the basic functionality at the core and building optional and developer-assist functionality (debug, etc.) in outer layers.
- Full HLSL integration - All Direct3D 10 shaders are written in HLSL and implemented with the common shader core.

- An increase in the number of render targets, textures, and samplers. There is also no shader length limit.
- Full support for:
  - Integer and bitwise shader operations
  - Readback of depth/stencil and multisampled resources in the shader
  - Multisample Alpha-to-Coverage

There are additional behavioral differences that Direct3D 9 developers should also be aware of. For a more complete list, refer to [Direct3D 9 to Direct3D 10 Considerations \(Direct3D 10\)](#) [<http://msdn2.microsoft.com/en-us/library/bb205073.aspx>].

# State Objects (Direct3D 10)

In Direct3D 10, device state is grouped into state objects which greatly reduce the cost of state changes. There are several state objects, and each one is designed to initialize a set of state for a particular pipeline stage. You can create up to 4096 of each type of state object.

## Input-Layout State

This group of state (see `D3D10_INPUT_ELEMENT_DESC`) dictates how the input assembler stage reads data out of the input buffers and assembles it for use by the vertex shader. This includes state such as the number of elements in the input buffer and the signature of the input data. The input-assembler stage is a new stage in the pipeline whose job is to stream primitives from memory into the pipeline.

To create a input-layout-state object, see `ID3D10Device::CreateInputLayout`.

## Rasterizer State

This group of state (see `D3D10_RASTERIZER_DESC`) initializes the rasterizer stage. This object includes state such as fill or cull modes, enabling a scissor rectangle for clipping, and setting multisample parameters. This stage rasterizes primitives into pixels, performing operations like clipping and mapping primitives to the viewport.

To create a rasterizer-state object, see `ID3D10Device::CreateRasterizerState`.

## Depth-Stencil State

This group of state (see `D3D10_DEPTH_STENCIL_DESC`) initializes the depth-stencil portion of the output-merger stage. More specifically, this object initializes depth and stencil testing.

To create a depth-stencil-state object, see `ID3D10Device::CreateDepthStencilState`.

## Blend State

This group of state (see `D3D10_BLEND_DESC`) initializes the blending portion of the output-merger stage.

To create a blend-state object, see `ID3D10Device::CreateBlendState`.

## Sampler State

This group of state (see `D3D10_SAMPLER_DESC`) initializes a sampler object. A sampler object is used by the shader stages to filter textures in memory.

Differences between Direct3D 9 and Direct3D 10:

In Direct3D 10, the sampler object is no longer bound to a specific texture - it just describes how to do filtering given any attached resource.

To create a sampler-state object, see `ID3D10Device::CreateSamplerState`.

## Performance Considerations

Designing the API to use state objects creates several performance advantages. These include validating state at object creation time, enabling caching of state objects in hardware, and greatly reducing the amount of state that is passed during a state-setting API call (by passing a handle to the state object instead of the state).

To achieve these performance improvements, you should create your state objects when your application starts up, well before your render loop. State objects are immutable, that is, once they are created, you cannot change them. Instead you must destroy and recreate them. To cope with this restriction, you can create up to 4096 of each type of state objects. For example, you could create several sampler objects with various sampler-state combinations. Changing the sampler state is then accomplished by calling the appropriate Set API which passes a handle to the object (as opposed to the sampler state). This significantly reduces the amount of overhead during each render frame for changing state since the number of calls and the amount of data are greatly reduced.

Alternatively, you can choose to use the effect system which will automatically manage efficient creation and destruction of state objects for your application.

# API Layers (Direct3D 10)

The Direct3D 10 runtime is constructed with layers, starting with the basic functionality at the core and building optional and developer-assist functionality in outer layers.

## Core Layer

The core layer exists by default; providing a very thin mapping between the API and the device driver, minimizing overhead for high-frequency calls. As the core layer is essential for performance, it only performs critical validation.

The remaining layers are optional. As a general rule, layers add functionality, but do not modify existing behavior. For example, core functions will have the same return values independent of the debug layer being instantiated, although additional debug output may be provided if the debug layer is instantiated.

Create layers when a device is created by calling D3D10CreateDevice and supplying one or more D3D10\_CREATE\_DEVICE\_FLAG values.

## Debug Layer

The debug layer provides extensive additional parameter and consistency validation (such as validating shader linkage and resource binding, validating parameter consistency, and reporting error descriptions). The output generated by the debug layer consists of a queue of strings which are accessible using the ID3D10InfoQueue Interface (see [Customize Debug Output with ID3D10InfoQueue \(Direct3D 10\)](#)). Errors generated by the core layer are highlighted with warnings by the debug layer.

To create a device that supports the debug layer, you must install the DirectX SDK (to get D3D10SDKLayers.DLL), and then specify the D3D10\_CREATE\_DEVICE\_DEBUG flag when calling D3D10CreateDevice. Of course, running an application with the debug layer will be substantially slower.

## Switch-to-Reference Layer

This layer enables an application to transition between a hardware device (HAL) and a reference or software device (REF). To switch a device, you must first create a device that supports the switch-to-reference layer (see D3D10CreateDevice) and then call ID3D10SwitchToRef::SetUseRef.

All device state, resources and objects are maintained through this device transition. However, it is sometimes not possible to exactly match resource data; this is especially true with multisampled resources. This is due to the fact that some information is available to a HAL device that is not available to a REF device.

Virtually all real-time applications use the HAL implementation of the pipeline. When the pipeline is switched from a hardware device to a reference device, rendering operations are done simultaneously in both hardware and software. As the software device is rendering, it will require that resources are downloaded to system memory. This may require other resources cached in system memory to be evicted to make room. In general, this is not a problem except in the case of multisampled resources.

Since multisampling can be hardware specific it can be difficult to match exactly the results of multisampled resources between a HAL and REF implementation.

## Thread-Safe Layer

This layer is designed to allow multi-threaded applications to access the device from multiple threads.

Direct3D 10 enables an application to exercise explicit control over the device synchronization primitive with device functions that can be invoked at any time over the lifetime of the device, including enabling and disabling the use of the critical section (temporarily enabling/disabling multithread protection), and a means to take and release the critical section lock and thereby hold the lock over multiple Direct3D 10 API entrypoints.

This layer is enabled by default, but if not present has no performance impact on single-thread accessed devices.

Differences between Direct3D 9 and Direct3D 10:

Unlike Direct3D 9, the Direct3D 10 API defaults to fully thread-safe.

# Customize Debug Output with ID3D10InfoQueue (Direct3D 10)

The information queue is managed by an interface (see [ID3D10InfoQueue Interface](#)) that stores, retrieves, and filters debug messages. The queue consists of: a message queue, an optional storage filter stack, and a optional retrieval filter stack. The storage-filter stack can be used to filter the messages you want stored; the retrieval-filter stack can be used to filter the messages you want stored. Once you have filtered a message, the message will be printed out to the debug window and stored in the appropriate stack.

In general:

Call `ID3D10InfoQueue::AddApplicationMessage` to generate user-defined messages

Call `ID3D10InfoQueue::GetMessage` is used to get messages (that pass an optional retrieval filter).

## Registry Controls

Use registry keys to adjust filter settings, adjust break points, and to mute the debug output. The debug layer will check these paths for registry keys; the first path found will be used.

1. `HKCU\Software\Microsoft\Direct3D\<user-defined subkey>`
2. `HKLM\Software\Microsoft\Direct3D\<user-defined subkey>`
3. `HKCU\Software\Microsoft\Direct3D`

Where:

- `HKCU` stand for `HKEY_CURRENT_USER`, and `HKLM` stand for `HKEY_LOCAL_MACHINE`.
- `user-defined subkey` is an arbitrary name to store debug settings for an application

## Filtering Debug Messages using Registry Keys

If the registry contains an `InfoQueueStorageFilterOverride` key (and is non-zero), then messages (and debug output) can be filtered by adding the following registry controls.

- `DWORD Mute_CATEGORY_*` - Debug output if this key is non-zero.
- `DWORD Mute_SEVERITY_*` - Debug output is disabled if this key is non-zero.
- `DWORD Mute_ID_*` - Message name or number can be used for \* (just like for `BreakOn_ID_*` described earlier). Debug output is disabled if this key is non-zero.

- DWORD Unmute\_SEVERITY\_INFO - Debug output is ENABLED if this key is non-zero. By default when InfoQueueStorageFilterOverride is enabled, debug messages with severity INFO are muted - therefore for this key allows INFO to be turned back on.

These controls change whether a message is recorded or displayed; they do not affect whether an API passes or fails.

### **Setting Break Conditions using Registry Keys**

Applications can be forced to break on a message using the following registry keys.

EnableBreakOnMessage - This key enables breaking on messages (and causes InfoQueue's SetBreakOnCategory()/SetBreakOnSeverity()/SetBreakOnID() settings to be ignored). The actual messages to break on are defined using one or more BreakOn\_\* values defined below.

- BreakOn\_CATEGORY\_\* - Break on any message passing through the storage filters. \* is one of the D3D10\_MESSAGE\_CATEGORY messages.
- BreakOn\_SEVERITY\_\* - Break on any message passing through the storage filters. \* is one of the D3D10\_MESSAGE\_SEVERITY\_messages.
- BreakOn\_ID\_\* - Break on any message passing through the storage filters. \* is one of the D3D10\_MESSAGE\_ID\_messages or can be the numerical value of the error enum. For example, Suppose the message with ID "D3D10\_MESSAGE\_ID\_HYPOTHETICAL" had the value 123 in the D3D10\_MESSAGE\_ID enum. In this case, creating the value BreakOn\_ID\_HYPOTHETICAL=1 or BreakOn\_ID\_123=1 would both accomplish the same thing - break when a message having ID D3D10\_MESSAGE\_ID\_HYPOTHETICAL is encountered.

### **Muting Debug Output using Registry Keys**

Debug output can be muted using a MuteDebugOutput key. The presence of this value in the registry forces override of the InfoQueue's ID3D10InfoQueue::SetMuteDebugOutput method.

MuteDebugOutput stops messages that pass the storage filter from being sent to debug output.

# Reference Counting (Direct3D 10)

Direct3D10 pipeline Set functions do not hold a reference to the DeviceChild objects. This means that each application must hold a reference to the DeviceChild object for as long as the object needs to be bound to the pipeline. When the reference count of an object drops to zero, the object will be unbound from the pipeline and destroyed. This style of reference holding is also known as weak-reference holding because each pipeline binding location holds a weak reference to the interface/object that is bound to it.

For example:

```
pDevice->CreateRasterizerState( ..., &pRasterizerState );
pDevice->RSSetState( pRasterizerState );

pDevice->RSGetState( &pCurRasterizerState );
// pCurRasterizerState will be equal to pRasterizerState.
pCurRasterizerState->Release();

pRasterizerState->Release();
// Since app released the final ref on this object, it is unbound.
pDevice->GetRasterizerState( &pCurRasterizerState );
// pCurRasterizerState will be equal to NULL.
```

Differences between Direct3D 9 and Direct3D 10:

In Direct3D 9, pipeline Set functions hold a reference to the device's objects; in Direct3D10 pipeline Set functions do not hold a reference to the DeviceChild objects.

# Pipeline Stages (Direct3D 10)

The Direct3D 10 programmable pipeline is designed for generating graphics for realtime gaming applications. The conceptual diagram below illustrates the data flow from input to output through each of the programmable stages.

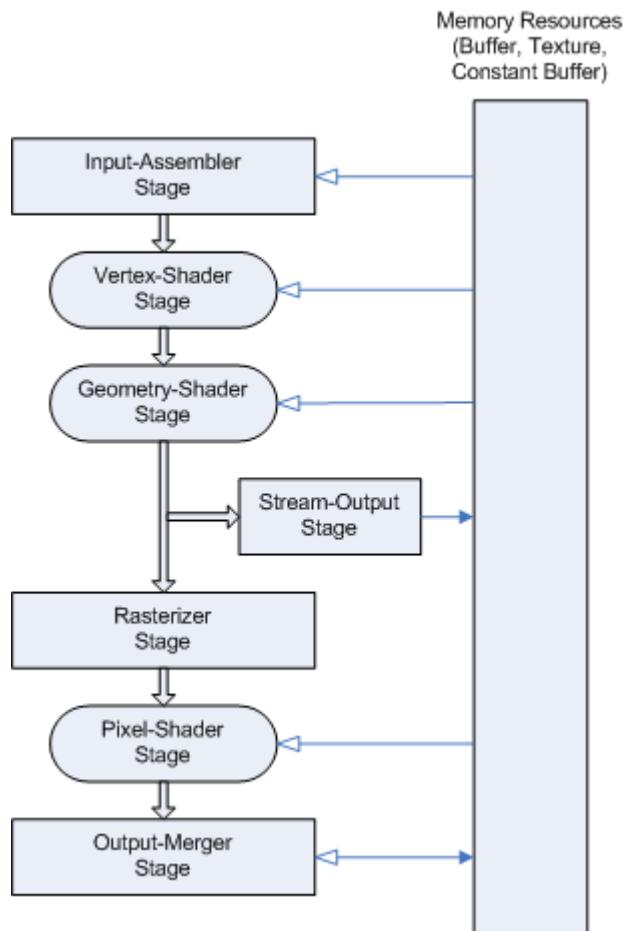


Figure 1. Direct3D 10 Pipeline Stages

All of the stages are configurable via the Direct3D 10 API. Stages featuring common shader cores (the rounded rectangular blocks) are programmable using the HLSL programming language. As you will see, this makes the pipeline extremely flexible and adaptable. The purpose of each of the stages is listed below.

- **Input Assembler Stage** - The input assembler stage is responsible for supplying data (triangles, lines and points) to the pipeline.
- **Vertex Shader Stage** - The vertex shader stage processes vertices, typically performing operations such as transformations, skinning, and lighting. A vertex shader always takes a single input vertex and produces a single output vertex.

- Geometry Shader Stage - The geometry shader processes entire primitives. Its input is a full primitive (which is three vertices for a triangle, two vertices for a line, or a single vertex for a point). In addition, each primitive can also include the vertex data for any edge-adjacent primitives. This could include at most an additional three vertices for a triangle or an additional two vertices for a line. The Geometry Shader also supports limited geometry amplification and de-amplification. Given an input primitive, the Geometry Shader can discard the primitive, or emit one or more new primitives.
- Stream Output Stage - The stream output stage is designed for streaming primitive data from the pipeline to memory on its way to the rasterizer. Data can be streamed out and/or passed into the rasterizer. Data streamed out to memory can be recirculated back into the pipeline as input data or read-back from the CPU.
- Rasterizer Stage - The rasterizer is responsible for clipping primitives, preparing primitives for the pixel shader and determining how to invoke pixel shaders.
- Pixel Shader Stage - The pixel shader stage receives interpolated data for a primitive and generates per-pixel data such as color.
- Output Merger Stage - The output merger stage is responsible for combining various types of output data (pixel shader values, depth and stencil information) with the contents of the render target and depth/stencil buffers to generate the final pipeline result.

# Input-Assembler Stage (Direct3D 10)

The Direct3D 10 API separates functional areas of the pipeline into stages; the first stage in the pipeline is the input-assembler (IA) stage.

The purpose of the input-assembler stage is to read primitive data (points, lines and/or triangles) from user-filled buffers and assemble the data into primitives that will be used by the other pipeline stages. The IA stage can assemble vertices into several different primitive types (such as line lists, triangle strips, or primitives with adjacency). New primitive types (such as a line list with adjacency or a triangle list with adjacency) have been added to support the geometry shader.

While assembling primitives, a secondary purpose of the IA is to attach system-generated values to help make shaders more efficient. System-generated values are text strings that are also called semantics. All three shader stages are constructed from a common shader core, and the shader core uses system-generated values (such as a primitive id, an instance id, or a vertex id) so that a shader stage can reduce processing to only those primitives, instances, or vertices that have not already been processed.

As shown in the pipeline block diagram, once the IA stage reads data from memory (assembles the data into primitives and attaches system-generated values), the data is output to the vertex shader stage.

## Getting Started with the Input-Assembler Stage (Direct3D 10)

There are a few steps necessary to initialize the input-assembler stage. For example, you need to create buffer resources with the vertex data that the pipeline needs, tell the IA stage where the buffers are and what type of data they contain, and specify the type of primitives to assemble from the data. Without getting into generation of system-level values yet, the basic steps involved in setting up the IA stage are covered in this topic:

- Create Input Buffers - Create and initialize input buffers with input vertex data.
- Create the Input-Layout Object - Define how the vertex buffer data will be streamed into the IA stage using an input-layout object.
- Binding Objects to the Input-Assembler Stage - Bind the created objects (input buffers and the input-layout object) to the IA stage.
- Specify the Primitive Type - Identify how the vertices will be assembled into primitives.
- Draw APIs - Send the data bound to the IA stage through the pipeline.

## Create Input Buffers

There are two types of input buffers: vertex buffers and index buffers. Vertex buffers supply vertex data to the IA stage. Index buffers are optional; they provide indices to vertices from the vertex buffer. You may create one or more vertex buffers, and optionally an index buffer. For help getting started creating these input resources, see the following guides:

- Resources - Create a Vertex Buffer
- Resources - Create an Index Buffer

Once the buffer resources are created, you need to create an input-layout object to describe the data layout to the IA stage, and then you need to bind the buffer resources to the IA stage.

Creating and binding buffers is not necessary if your shaders doesn't buffers. Here is an example of a simple vertex and pixel shader that draw a single triangle ([see Using the Input-Assembler Stage without Buffers \(Direct3D 10\)](#)).

## Create the Input-Layout Object

The input-layout object encapsulates the input state of the IA stage. This includes a description of the input data that is bound to the IA stage. The data is streamed into the IA stage from memory, from one or more vertex buffers. The description identifies the input data that is bound from one or more vertex buffers and gives the runtime the ability to type check the input data types against the shader input parameter types. This type checking not only verifies that the types are compatible, but also that each of the elements that the shader requires is available in the buffer resource(s).

An input-layout object is created from an array of input-element descriptions as well as a pointer to the compiled shader (see `ID3D10Device::CreateInputLayout`). The array contains one or more input elements; each input element describes a single vertex-data element from a single vertex buffer. The entire description describes all the vertex-data elements from all the vertex buffers that will be bound to the IA stage. For example, the following layout description describes a single vertex buffer that contains three vertex-data elements:

```
D3D10_INPUT_ELEMENT_DESC layout[] =
{
    { L"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
        D3D10_INPUT_PER_VERTEX_DATA, 0 },
    { L"TEXTURE0", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 12,
        D3D10_INPUT_PER_VERTEX_DATA, 0 },
    { L"NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 1, 20,
        D3D10_INPUT_PER_VERTEX_DATA, 0 },
};
```

Each input-element description describes a single element of vertex-buffer data, including its size, type, location, and purpose. This declaration contains two elements, corresponding to two rows. Each row is

enclosed in curly braces to make it easier to read. The first row or element declares per-vertex position data, the second row or element declares per-vertex color data.

Each row identifies the next element to be read from a vertex buffer. In each row, you can identify the type of data by using the semantic, the semantic index, and the data format. A semantic is a text string that identifies how the data will be used. In this example, the first row identifies 3-component position data (XYZ, for example); the second row identifies 2-component texture data (UV, for example); and the third row identifies 3-component color data (RGB, for example).

In this example description, the semantic index (which is the second parameter) is set to zero for all three rows. The semantic index helps distinguish between two rows that use the same semantics. Since there are no similar semantics in this example declaration, the semantic index can be set to its default value, zero.

The third parameter is the format. The format (see DXGI\_FORMAT) specifies the number of components per element, and the data type, which defines the size of the data for each element. The format can be fully typed at resource creation time, or you may create a resource using a typeless format, which identifies the number of components in an element, but leaves the data type undefined.

## Input Slots

Data enters the IA stage through inputs called input slots. The IA stage has n input slots, which are designed to accommodate up to n vertex buffers that provide input data. Each vertex buffer must be assigned to a different slot; this information is stored in the input-layout declaration when the input-layout object is created. Since the slots are zero-based, the first slot is slot 0. You may also specify an offset, from the start of each buffer to the first element in the buffer to be read.

The next two parameters are the input slot and the input offset. The input slot is the number IA stage input. When you use multiple buffers, you can bind them to one or more input slots. The input offset is the number of bytes between the start of the buffer and the beginning of the data.

## Reusing Input-Layout Objects

Each input-layout object is created based on a shader signature; this allows the API to validate the input-layout-object elements against the shader-input signature to make sure there is an exact match of types and semantics. You can create a single input-layout object for many shaders, as long as all the shader-input signatures exactly match.

## Binding Objects to the Input-Assembler Stage

After creating vertex buffer resources and an input layout object, they can be bound to the IA stage by calling ID3D10Device::IASetVertexBuffers and ID3D10Device::IASetInputLayout. The following example shows the binding of a single vertex buffer and an input-layout object to the IA stage:

```
UINT stride = sizeof( SimpleVertex );
UINT offset = 0;
g_pd3dDevice->IASetVertexBuffers(
    0, // the first input slot for binding
```

```

        1,           // the number of buffers in the array
        &g_pVertexBuffer, // the array of vertex buffers
        &stride,         // array of stride values, one for each buffer
        &offset );       // array of offset values, one for each buffer

// Set the input layout
g_pd3dDevice->IASetInputLayout( g_pVertexLayout );

```

Binding the input-layout object only requires a pointer to the object.

In the preceding example, a single vertex buffer was bound; however, multiple vertex buffers can be bound with a single call to ID3D10Device::IASetVertexBuffers, and the following code shows such a call to bind three vertex buffers:

```

UINT strides[3];
strides[0] = sizeof(SimpleVertex1);
strides[1] = sizeof(SimpleVertex2);
strides[2] = sizeof(SimpleVertex3);
UINT offsets[3] = { 0, 0, 0 };
g_pd3dDevice->IASetVertexBuffers(
    0,           //first input slot for binding
    3,           //number of buffers in the array
    &g_pVertexBuffer, //array of three vertex buffers
    &strides,      //array of stride values, one for each buffer
    &offsets );     //array of offset values, one for each buffer

```

An index buffer can be bound to the IA stage by calling ID3D10Device::IASetIndexBuffer.

## Specify the Primitive Type

After the input buffers have been bound, the IA stage must also know how to assemble the vertices into primitives. This is done by specifying the primitive type with ID3D10Device::IASetPrimitiveTopology. The following code calls this function to define the data as a triangle list without adjacency:

```
g_pd3dDevice->IASetPrimitiveTopology( D3D10_PRIMITIVE_TOPOLOGY_TRIANGLELIST );

```

The rest of the primitive types are listed in D3D10\_PRIMITIVE\_TOPOLOGY.

## Draw APIs

After input resources have been bound to the pipeline, an application calls a draw API to render primitives. There are several draw APIs, which are shown in the following table; some use index buffers, some use instance data, and some reuse data from the streaming-output stage as input to the input-assembler stage.

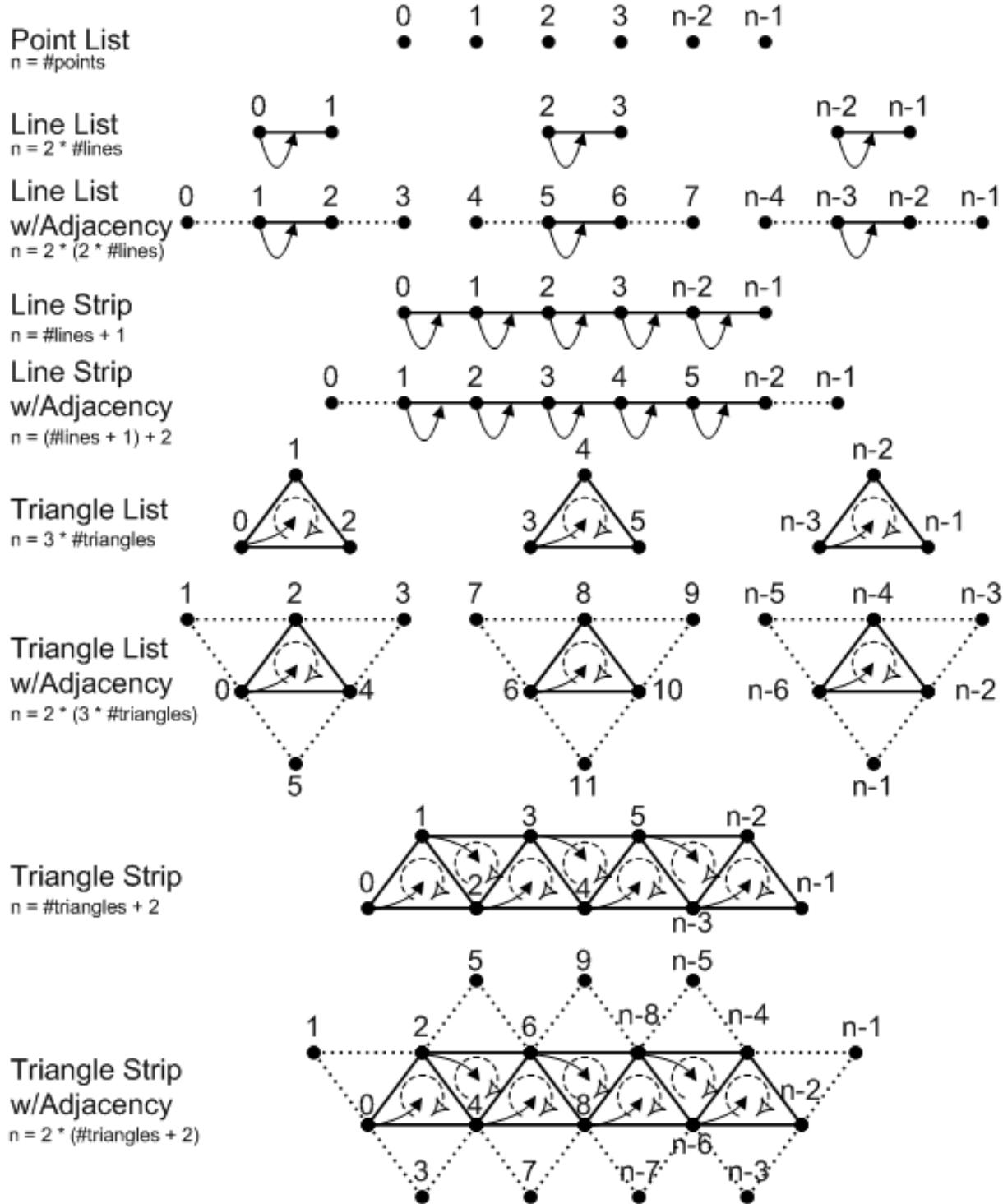
| <b>Draw API</b>                    | <b>Description</b>                                                                                      |
|------------------------------------|---------------------------------------------------------------------------------------------------------|
| ID3D10Device::Draw                 | Draws non-indexed, non-instanced primitives.                                                            |
| ID3D10Device::DrawInstanced        | Draws non-indexed, instanced primitives.                                                                |
| ID3D10Device::DrawIndexed          | Draws indexed, non-instanced primitives.                                                                |
| ID3D10Device::DrawIndexedInstanced | Draws indexed, instanced primitives.                                                                    |
| ID3D10Device::DrawAuto             | Draws non-indexed, non-instanced primitives from input data that comes from the streaming-output stage. |

# Primitive Types (Direct3D 10)

The input-assembler stage reads data from vertex and index buffers, assembles the data into primitives, and then sends the data to the remaining pipeline stages. The IA uses three pieces of information to fully specify primitives: a vertex, a winding direction, and a leading vertex.

| Symbol                                                                            | Name               | Description                                                                                                                                                                                         |
|-----------------------------------------------------------------------------------|--------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ●                                                                                 | Vertex             | A point in 3D space.                                                                                                                                                                                |
|  | Winding Direction  | Indicates the vertex order when assembling a primitive. The winding direction can be either clockwise or counterclockwise; specify this to Direct3D by calling ID3D10Device::CreateRasterizerState. |
|  | Leading Vertex     | The first vertex in a sequence of vertices.                                                                                                                                                         |
| n                                                                                 | Number of Vertices | The number of vertices in the assembled primitive.                                                                                                                                                  |

The following diagram shows how these three pieces of information can be combined to assemble the primitive types supported by Direct3D.



Use `ID3D10Device::IASetPrimitiveTopology` to specify the primitive type of the data that will be streamed into the input-assembler stage.

# Using the Input-Assembler Stage without Buffers (Direct3D 10)

Creating and binding buffers is not necessary if your shaders doesn't buffers. Here is an example of a simple vertex and pixel shader that draw a single triangle.

## Vertex Shader

For example, you could declare a vertex shader that creates its own vertices.

```
struct VSIn
{
    uint vertexId : SV_VertexID;
};

struct VSOut
{
    float4 pos : SV_Position;
    float4 color : color;
};

VSOut VSmain(VSIn input)
{
    VSOut output;

    if (input.vertexId == 0)
        output.pos = float4(0.0, 0.5, 0.5, 1.0);
    else if (input.vertexId == 2)
        output.pos = float4(0.5, -0.5, 0.5, 1.0);
    else if (input.vertexId == 1)
        output.pos = float4(-0.5, -0.5, 0.5, 1.0);

    output.color = clamp(output.pos, 0, 1);

    return output;
}
```

## Pixel Shader Code

```
// NoBuffer.fx
// Copyright (c) 2004 Microsoft Corporation. All rights reserved.
//

struct PSIn
{
    float4 pos : SV_Position;
    linear float4 color : color;
};
```

```

struct PSOut
{
    float4 color : SV_Target;
};

PSOut PSmain(PSIn input)
{
    PSOut output;

    output.color = input.color;

    return output;
}

```

## Technique

```

VertexShader vsCompiled = CompileShader( vs_4_0, VSmain() );

technique10 t0
{
    pass p0
    {
        SetVertexShader( vsCompiled );
        SetGeometryShader( NULL );
        SetPixelShader( CompileShader( ps_4_0, PSmain() ) );
    }
}

```

## Application Code

```

m_pD3D10Device->IASetInputLayout( NULL );

m_pD3D10Device->IASetPrimitiveTopology( D3D10_PRIMITIVE_TOPOLOGY_TRIANGLELIST
) ;

ID3D10EffectTechnique * pTech = NULL;
pTech = m_pEffect->GetTechniqueByIndex(0);
pTech->GetPassByIndex(iPass)->Apply(0);

m_pD3D10Device->Draw( 3, 0 );

```

# Using System-Generated Values

System-generated values are generated by the IA stage (based on user-supplied input semantics) to allow certain efficiencies in shader operations. By attaching data, such as an instance id (visible to VS), a vertex id (visible to VS), or a primitive id (visible to GS/PS), a subsequent shader stage may look for these system values to optimize processing in that stage. For instance, the VS stage may look for the instance id to grab additional per-vertex data for the shader or to perform other operations; the GS and PS stages may use the primitive id to grab per-primitive data in the same way.

## VertexID

A VertexID is used by each shader stage to identify each vertex. It is a 32-bit unsigned integer whose default value is 0. It is assigned to a vertex when the primitive is processed by the IA stage. Attach the vertex-id semantic to the shader input declaration to inform the IA stage to generate a per-vertex id.

The IA will add a vertex id to each vertex for use by shader stages. For each draw call, the vertex id is incremented by 1. Across indexed draw calls, the count resets back to the start value. For DrawIndexed and DrawIndexedInstanced, the vertex id represents the index value. If the vertex id overflows (exceeds 232), it wraps to 0.

For all primitive types, vertices have a vertex id associated with them (regardless of adjacency).

## PrimitiveID

A PrimitiveID is used by each shader stage to identify each primitive. It is a 32-bit unsigned integer whose default value is 0. It is assigned to a primitive when the primitive is processed by the IA stage. To inform the IA stage to generate a primitive id, attach the primitive-id semantic to the shader input declaration.

The IA stage will add a primitive id to each primitive for use by the geometry shader or the pixel shader stage (whichever is the first stage active after the IA stage). For each indexed draw call, the primitive id is incremented by 1, however, the primitive id resets to 0 whenever a new instance begins. All other draw calls do not change the value of the instance id. If the instance id overflows (exceeds 232), it wraps to 0.

The pixel shader stage does not have a separate input for a primitive id; however, any pixel shader input that specifies a primitive id uses a constant interpolation mode.

There is no support for automatically generating a primitive id for adjacent primitives. For primitive types with adjacency, such as a triangle strip with adjacency, a primitive id is only maintained for the interior primitives (the non-adjacent primitives), just like the set of primitives in a triangle strip without adjacency.

## InstanceID

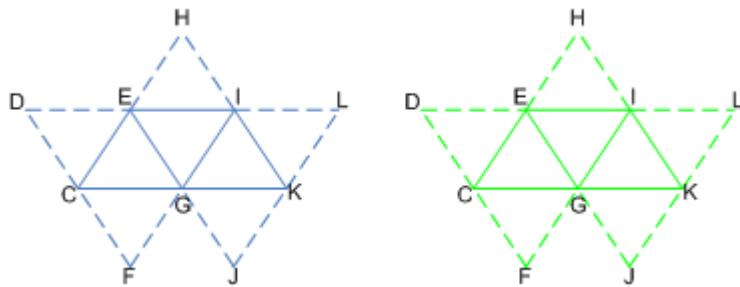
An InstanceID is used by each shader stage to identify the instance of the geometry that is currently being processed. It is a 32-bit unsigned integer whose default value is 0.

The IA stage will add an instance id to each vertex if the vertex shader input declaration includes the instance id semantic. For each indexed draw call, instance id is incremented by 1. All other draw calls do not change the value of instance id. If instance id overflows (exceeds 232), it wraps to 0.

## Example

The following example shows how system values are attached to an instanced triangle strip in the IA stage.

Figure 1. Generating System Values for Two Instances of a Triangle Strip



These tables show the system values generated for both instances of the same triangle strip. The first instance (instance U) is shown in blue, the second instance (instance V) is shown in green. The solid lines connect the vertices in the primitives, the dashed lines connect the adjacent vertices.

The following tables show the system-generated values for the instance U.

| Vertex Data | C,U | D,U | E,U | F,U | G,U | H,U | I,U | J,U | K,U | L,U |
|-------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| VertexID    | 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   |
| InstanceId  | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   |

| PrimitiveID | 0 | 1 | 2 |
|-------------|---|---|---|
| InstanceId  | 0 | 0 | 0 |

The following tables show the system-generated values for the instance V.

| Vertex Data | C,V | D,V | E,V | F,V | G,V | H,V | I,V | J,V | K,V | L,V |
|-------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| VertexID    | 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   |
| InstanceId  | 1   | 1   | 1   | 1   | 1   | 1   | 1   | 1   | 1   | 1   |

| PrimitiveID | 0 | 1 | 2 |
|-------------|---|---|---|
| InstanceId  | 1 | 1 | 1 |

The input assembler generates each of the ids (vertex, primitive, and instance). Notice also that each instance is given a unique instance id. The data ends with the strip cut, which separates each instance of the triangle strip.

# Shader Stages (Direct3D 10)

The Direct3D 10 pipeline contains 3 programmable-shader stages (shown as the rounded blocks in the pipeline functional diagram):

Each shader stage exposes its own unique functionality, built on the shader model 4.0 common shader core.

## Vertex-Shader Stage

The vertex-shader (VS) stage processes vertices from the input assembler, performing per-vertex operations such as transformations, skinning, morphing, and per-vertex lighting. Vertex shaders always operate on a single input vertex and produce a single output vertex. The vertex shader stage must always be active for the pipeline to execute. If no vertex modification or transformation is required, a pass-through vertex shader must be created and set to the pipeline.

Each vertex shader input vertex can be comprised of up to 16 32-bit vectors (up to 4 components each) and each output vertex can be comprised of as many as 16 32-bit 4-component vectors. All vertex shaders must have a minimum of one input and one output, which can be as little as one scalar value.

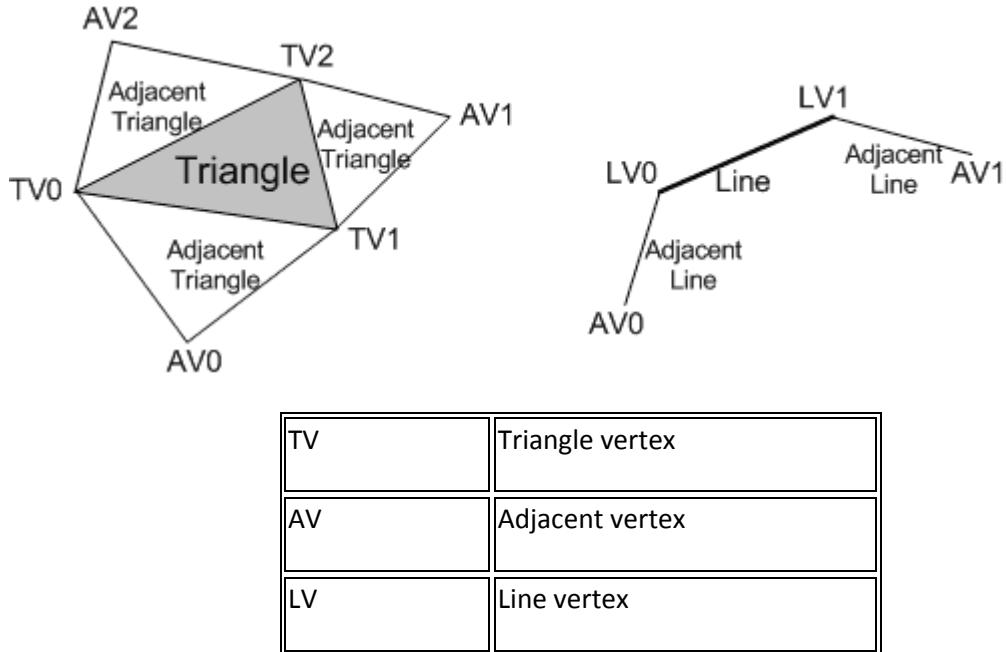
The vertex-shader stage can consume two system generated values from the input assembler: `VertexID` and `InstanceID` (see System Values and Semantics). Since `VertexID` and `InstanceID` are both meaningful at a vertex level, and IDs generated by hardware can only be fed into the first stage that understands them, these ID values can only be fed into the vertex-shader stage.

Vertex shaders are always run on all vertices, including adjacent vertices in input primitive topologies with adjacency. The number of times that the vertex shader has been executed can be queried from the CPU using the `VSIInvocations` pipeline statistic.

A vertex shader can perform load and texture sampling operations where screen-space derivatives are not required (using HLSL intrinsic functions `samplelevel`, `samplecmplevelzero`, `samplegrad`).

## Geometry-Shader Stage

The geometry-shader (GS) stage runs application-specified shader code with vertices as input and the ability to generate vertices on output. Unlike vertex shaders, which operate on a single vertex, the geometry shader's inputs are the vertices for a full primitive (two vertices for lines, three vertices for triangles, or single vertex for point). Geometry shaders can also bring in the vertex data for the edge-adjacent primitives as input (an additional two vertices for a line, an additional three for a triangle). The image below shows a triangle and a line with adjacent vertices.



The geometry-shader stage can consume the `SV_PrimitiveID` system-generated value that is auto-generated by the IA. This allows per-primitive data to be fetched or computed if desired.

The geometry-shader stage is capable of outputting multiple vertices forming a single selected topology (GS stage output topologies available are: tristrip, linestrip, and pointlist). The number of primitives emitted can vary freely within any invocation of the geometry shader, though the maximum number of vertices that could be emitted must be declared statically. Strip lengths emitted from a geometry shader invocation can be arbitrary, and new strips can be created via the `RestartStrip` HLSL intrinsic function.

Geometry shader output may be fed to the rasterizer stage and/or to a vertex buffer in memory via the stream output stage. Output fed to memory is expanded to individual point/line/triangle lists (exactly as they would be passed to the rasterizer).

When a geometry shader is active, it is invoked once for every primitive passed down or generated earlier in the pipeline. Each invocation of the geometry shader sees as input the data for the invoking primitive, whether that is a single point, a single line, or a single triangle. A triangle strip from earlier in the pipeline would result in an invocation of the geometry shader for each individual triangle in the strip (as if the strip were expanded out into a triangle list). All the input data for each vertex in the individual primitive is available (i.e. 3 vertices for triangle), plus adjacent vertex data if applicable/available.

A geometry shader outputs data one vertex at a time by appending vertices to an output stream object. The topology of the streams is determined by a fixed declaration, choosing one of: `PointStream`, `LineStream`, or `TriangleStream` as the output for the GS stage. There are three types of stream objects available, `PointStream`, `LineStream` and `TriangleStream` which are all templated objects. The topology of the output is determined by their respective object type, while the format of the vertices appended to

the stream is determined by the template type. Execution of a geometry shader instance is atomic from other invocations, except that data added to the streams is serial. The outputs of a given invocation of a geometry shader are independent of other invocations (though ordering is respected). A geometry shader generating triangle strips will start a new strip on every invocation.

When a geometry shader output is identified as a System Interpreted Value (e.g. `SV_RenderTargetArrayIndex` or `SV_Position`), hardware looks at this data and performs some behavior dependent on the value, in addition to being able to pass the data itself to the next shader stage for input. When such data output from the geometry shader has meaning to the hardware on a per-primitive basis (such as `SV_RenderTargetArrayIndex` or `SV_ViewportArrayIndex`), rather than on a per-vertex basis (such as `SV_ClipDistance[n]` or `SV_Position`), the per-primitive data is taken from the leading vertex emitted for the primitive.

Partially completed primitives could be generated by the geometry shader if the geometry shader ends and the primitive is incomplete. Incomplete primitives are silently discarded. This is similar to the way the IA treats partially completed primitives.

The geometry shader can perform load and texture sampling operations where screen-space derivatives are not required (`samplelevel`, `samplecmplevelzero`, `samplegrad`).

Algorithms that can be implemented in the geometry shader include:

- Point Sprite Expansion
- Dynamic Particle Systems
- Fur/Fin Generation
- Shadow Volume Generation
- Single Pass Render-to-Cubemap
- Per-Primitive Material Swapping

Per-Primitive Material Setup - Including generation of barycentric coordinates as primitive data so that a pixel shader can perform custom attribute interpolation (for an example of higher-order normal interpolation, see `CubeMapGS Sample`).

## Pixel-Shader Stage

The pixel-shader (PS) stage is invoked by the rasterizer stage, to calculate a per-pixel value for each pixel in a primitive that gets rendered. A pixel shader enables rich shading techniques such as per-pixel lighting and post-processing. A pixel shader is a program that combines constant variables, texture values, interpolated per-vertex values, and other data to produce per-pixel outputs. The stage preceding the rasterizer stage (GS stage or the VS stage if the geometry shader is NULL) must output vertex positions in homogenous clip space.

A pixel shader can input up to 32 32-bit 4-component data for the current pixel location. It is only when the geometry shader is active that all 32 inputs can be fed with data from above in the pipeline. In the absence of the geometry shader, only up to 16 4-component elements of data can be input from upstream in the pipeline.

Input data available to the pixel shader includes vertex attributes that can be chosen, on a per-element basis, to be interpolated with or without perspective correction, or be treated as per-primitive constants. In addition, declarations in a pixel shader can indicate which attributes to apply centroid evaluation rules to. Centroid evaluation is relevant only when multisampling is enabled, since cases arise where the pixel center may not be covered by the primitive (though subpixel center(s) are covered, hence causing the pixel shader to run once for the pixel). Attributes declared with centroid mode must be evaluated at a location covered by the primitive, preferably at a location as close as possible to the (non-covered) pixel center.

A pixel shader can output up to 8 32-bit 4-component data for the current pixel location to be combined with the render target(s), or no color (if the pixel is discarded). A pixel shader can also output an optional 32-bit float scalar depth value for the depth test (SV\_Depth).

For each primitive entering the rasterizer, the pixel shader is invoked once for each pixel covered by the primitive. When multisampling, the pixel shader is invoked once per covered pixel, though depth/stencil tests occur for each covered multisample, and multisamples that pass the tests are updated with the pixel shader output color(s).

If there is no geometry shader, the IA stage is capable of producing one scalar per-primitive system-generated value to the pixel shader, the SV\_PrimitiveID, which can be read as input to the pixel shader. The pixel shader can also retrieve the the SV\_IsFrontFace value, generated by the rasterizer stage.

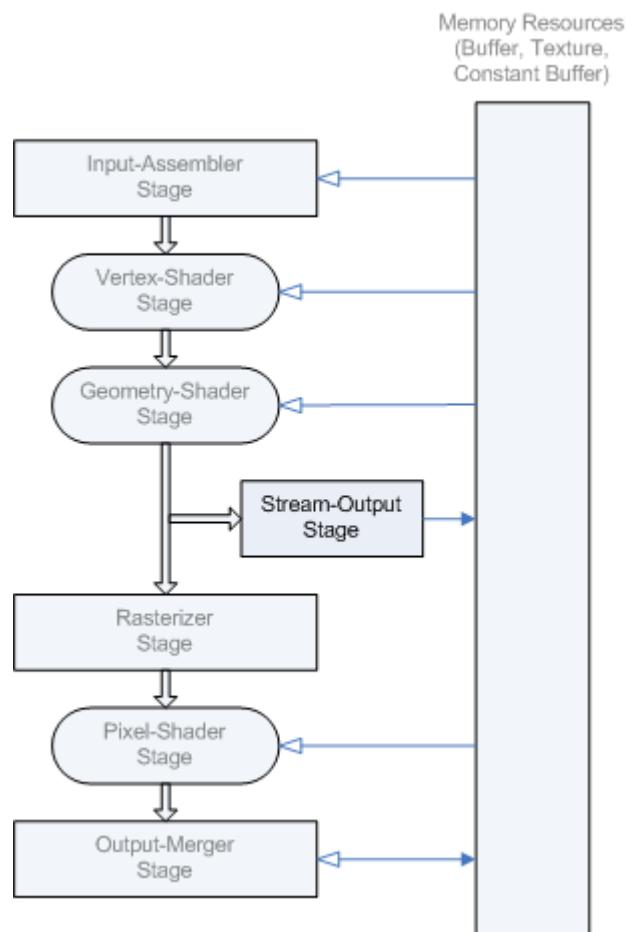
One of the inputs to the pixel shader can be declared with the name SV\_Position, which means it will be initialized with the pixel's float32 xyzw position. Note that w is the reciprocal of the linearly interpolated 1/w value. When the rendertarget is a multisample buffer or a standard rendertarget, the xy components of position contain pixel center coordinates (which have a fraction of 0.5f).

The pixel shader intrinsic functions produce or use derivatives of quantities with respect to screen space x and y. The most common use for derivatives is to compute level-of-detail calculations for texture sampling and in the case of anisotropic filtering, selecting samples along the axis of anisotropy. Typically, hardware implementations run a pixel shader on multiple pixels (for example a 2x2 grid) simultaneously, so that derivatives of quantities computed in the pixel shader can be reasonably approximated as deltas of the values at the same point of execution in adjacent pixels.

# Stream-Output Stage (Direct3D 10)

The stream-output stage (SO) is located in the pipeline right after the geometry-shader stage and just before the rasterization stage.

Figure 1. Stream-Output Stage



The purpose of the SO stage is to write vertex data streamed out of the GS stage (or the VS stage if the GS stage is inactive) to one or more buffer resources in memory. Data streamed out to memory can be read back into the pipeline in a subsequent rendering pass, or can be copied to a staging resource for readback to the CPU. Since variable amounts of data can be generated by a geometry shader, the amount of data streamed out can vary. The `ID3D10Device::DrawAuto` API allows this variable amount of data to be processed in a subsequent pass without the need to query (the GPU) about the amount of data written to stream output.

When primitive types with adjacency are used as inputs to the geometry shader, the adjacent vertices are discarded and not streamed out.

When a triangle or line strip is bound to the input assembler, the strips are always converted into lists before they are streamed out. Vertices are always written out as complete primitives (e.g. 3 vertices at a time for triangles); incomplete primitives are never streamed out.

There are two primary ways streaming-output data can be fed back into the pipeline. One is to generate data that can be fed back into the input assembler, and the other is to generate data that can be read into shaders using load functions. If only one buffer is bound, the buffer can capture up to 64 scalar components of per-vertex data, as long as the total amount of data being output per-vertex is 256 bytes or less. Vertex stride can be up to 2048 bytes. If more than one buffer is bound, each can only capture a single element of per-vertex data. Stream output can write to up to 4 buffers simultaneously. If multiple buffers are bound and have different sizes, as soon as one of the buffers can no longer hold any more complete primitives, writing to all buffers is stopped.

## Getting Started with the Stream-Output Stage (Direct3D 10)

These are the steps required to initialize and execute the stream output stage:

### Compile a Geometry Shader

Given the following geometry shader (from Tutorial13):

```
struct GS_PPS_INPUT
{
    float4 Pos : SV_POSITION;
    float3 Norm : TEXCOORD0;
    float2 Tex : TEXCOORD1;
};

[maxvertexcount(3)]
void GS( triangle GS_PPS_INPUT input[3], inout TriangleStream<GS_PPS_INPUT>
TriStream )
{
    GS_PPS_INPUT output;

    //
    // Calculate the face normal
    //
    float3 faceEdgeA = input[1].Pos - input[0].Pos;
    float3 faceEdgeB = input[2].Pos - input[0].Pos;
    float3 faceNormal = normalize( cross(faceEdgeA, faceEdgeB) );

    for( int v=0; v<3; v++ )
    {
        output.Pos = input[v].Pos + float4(faceNormal*Explode, 0);
        output.Pos = mul( output.Pos, View );
    }
}
```

```

        output.Pos = mul( output.Pos, Projection );

        output.Norm = input[v].Norm;

        output.Tex = input[v].Tex;

        TriStream.Append( output );
    }

    TriStream.RestartStrip();
}

```

This shader calculates a face normal for each triangle, and outputs position, normal and texture coordinate data. A geometry shader looks just like a vertex or pixel shader, with the following exceptions:

- GS function return type - The function return type does one thing, declares the maximum number of vertices that can be output by the shader. In this case,

```
maxvertexcount[3]
```

defines the output to be a maximum of 3 vertices.

- GS input parameter declarations - This function takes two input parameters:

```
triangle GSPS_INPUT input[3] , inout TriangleStream<GSPS_INPUT>
TriStream
```

The first parameter is an array of vertices (3 in this case) defined by a `GSPS_INPUT` structure (which defines per-vertex data as a position, a normal and a texture coordinate). The first parameter also uses the `triangle` keyword, which means the input assembler stage must output data to the geometry shader as one of the triangle primitive types (triangle list or triangle strip).

The second parameter is a triangle stream defined by the type `TriangleStream<GSPS_INPUT>`. This means the parameter is an array of triangles, each of which is made up of three vertices (that contain the data from the members of `GSPS_INPUT`).

Use the `triangle` and `trianglestream` keywords to identify individual triangles or a stream of triangles in a GS.

- GS intrinsic function - The lines of code in the shader function use common-shader-core HLSL intrinsic functions except the last two lines, which call `Append` and `RestartStrip`. These functions are only available to a geometry shader. `Append` informs the geometry shader to append the output to the current strip; `RestartStrip` creates a new primitive strip. A new strip is implicitly created in every invocation of the GS stage.

The rest of the shader looks very similar to a vertex or pixel shader. The geometry shader uses a structure to declare input parameters and marks the position member with the `SV_POSITION` semantic

to tell the hardware that this is position data. The input structure identifies the other two input parameters as texture coordinates (even though one of them will contain a face normal). You could use your own custom semantic for the face normal if you prefer.

Having designed the geometry shader, call D3D10CompileShader to compile it like this:

```
DWORD dwShaderFlags = D3D10_SHADER_ENABLE_STRICTNESS;
ID3D10Blob** ppShader;

D3D10CompileShader( pSrcData, sizeof( pSrcData ),
    "Tutorial13.fx", NULL, NULL, "GS", "gs_4_0",
    dwShaderFlags, &ppShader, NULL );
```

Just like vertex and pixel shaders, you will need a shader flag to tell the compiler how you want the shader compiled (for debugging, optimized for speed etc...), the entry point function, and the shader model to validate against. This example creates a geometry shader built from the Tutorial13.fx file, by using the GS function. The shader is compiled for shader model 4.0.

## Create a Geometry-Shader Object with Stream Output

Once you know that you will be streaming the data from the geometry, and you have successfully compiled the shader, the next step is to call ID3D10Device::CreateGeometryShaderWithStreamOutput to create the geometry shader object.

But first, you need to declare the SO stage input signature. This signature matches or validates the GS outputs and the SO inputs at object creation time. Here's an example of the SO declaration:

```
D3D10_STREAM_OUTPUT_DECLARATION_ENTRY pDecl[] =
{
    // semantic name, semantic index, start component,
    // component count, output slot
    { L"SV_POSITION", 0, 0, 4, 0 }, // output all components of position
    { L"TEXCOORD0", 0, 0, 3, 0 }, // output the first 3 of the normal
    { L"TEXCOORD1", 0, 0, 2, 0 }, // output the first 2 texture coordinates
};

D3D10Device->CreateGeometryShaderWithStreamOut( pShaderBytecode, pDecl, 3,
    sizeof(pDecl), &pGS );
```

This function takes several parameters including:

- A pointer to the compiled geometry shader (or vertex shader if no geometry shader will be present and data will be streamed out directly from the VS). To get this pointer see Getting a Pointer to a Compiled Shader.
- A pointer to an array of declarations that describe the input data for the stream output stage. See D3D10\_SO\_DECLARATION\_ENTRY. You can supply up to 64 declarations, one for each different type of element to be output from the SO stage. The array of declaration entries

describes the data layout regardless of whether only a single Buffer or multiple buffers are to be bound for stream output.

- The number of elements that are written out by the SO stage.
- A pointer to the geometry shader object created (see ID3D10GeometryShader Interface).

The stream output declaration defines the way data is written to a buffer resource. You can add as many components as you want to the output declaration. Use the SO stage to write to a single buffer resource or many buffer resources. For a single buffer, the SO stage can write many different elements per-vertex. For multiple buffers, the SO stage can only write a single element of per-vertex data to each buffer.

To use the SO stage without using a geometry shader, call

ID3D10Device::CreateGeometryShaderWithStreamOutput and pass NULL in the pShaderBytecode parameter.

## Set the Output Targets

The last step is to set the SO stage buffers. Data can be streamed out into one or more buffers in memory for use later. This example shows how to create a single buffer that can be used for vertex data as well as for the SO stage to stream data into:

```
ID3D10Buffer *m_pBuffer;
int m_nBufferSize = 1000000;

D3D10_BUFFER_DESC bufferDesc =
{
    m_nBufferSize,
    D3D10_USAGE_DEFAULT,
    D3D10_BIND_STREAM_OUTPUT,
    0,
    0
};

D3D10Device->CreateBuffer( &bufferDesc, NULL, &m_pBuffer );
```

Create a buffer by calling ID3D10Device::CreateBuffer. This example illustrates default usage which is typical for a buffer resource that is expected to be updated fairly frequently by the CPU. The binding flag identifies the pipeline stage that the resource can be bound to. Any resource used by the SO stage must also be created with the D3D10\_BIND\_STREAM\_OUTPUT bind flag.

Once the buffer is successfully created, set it to the current device by calling ID3D10Device::SOSetTargets:

```
UINT offset[1] = 0;
D3D10Device->SOSetTargets( 1, &m_pBuffer, offset );
```

This call takes the number of buffers, a pointer to the buffers, and an array of offsets (one offset into each of the buffers that indicates where to begin writing data). Data will be written to these streaming-output buffers when a Draw command is issued. An internal variable keeps track of the position for where to begin writing data to the streaming-output buffers, and that variable will continue to increment until SOSetTargets is called again and a new offset value is specified.

All data written out to the target buffers will be 32-bit values.

# Rasterizer Stage (Direct3D 10)

This stage rasterizes primitives into pixels, interpolating specified vertex values across the primitive. To do so, the stage clips vertices to the view frustum, sets up the primitives for mapping to the 2D viewport, and determines how to invoke the pixel shader, if one is currently set to the pipeline. Some of these features are optional (like pixel shaders), however, the rasterizer always performs clipping, a perspective divide to transform the points into homogenous space, and maps the vertices to the viewport.

Vertices ( $x, y, z, w$ ), coming into the rasterizer stage are assumed to be in homogenous clip-space. In this coordinate space the X axis points right, Y points up and Z points away from camera.

You may disable rasterization by telling the pipeline there is no pixel shader (set the pixel shader stage to NULL with `ID3D10Device::PSSetShader`), and disabling depth and stencil testing (set `DepthEnable` and `StencilEnable` to FALSE in `D3D10_DEPTH_STENCIL_DESC`). While disabled, rasterization-related pipeline counters will not update.

On hardware that implements hierarchical Z-buffer optimizations, you may enable preloading the z-buffer by setting the pixel shader stage to NULL while enabling depth and stencil testing.

## Getting Started with the Rasterizer Stage (Direct3D 10)

The rasterizer stage is controlled with the following features:

### Set the Viewport

A viewport maps vertex positions (in clip space) into render target positions. This step scales the 3D positions into 2D space. A render target is oriented with the Y axes pointing down; this requires that the Y coordinates get flipped during the viewport scale. In addition, the x and y extents (range of the x and y values) are scaled to fit the viewport size according to the following formulas:

$$\begin{aligned}X &= (X + 1) \times \text{Viewport.Width} \times 0.5 + \text{Viewport.TopLeftX} \\Y &= (1 - Y) \times \text{Viewport.Height} \times 0.5 + \text{Viewport.TopLeftY} \\Z &= \text{Viewport.MinDepth} + Z * (\text{Viewport.MaxDepth} - \text{Viewport.MinDepth})\end{aligned}$$

Tutorial 1 creates a  $640 \times 480$  viewport using `D3D10_VIEWPORT` and by calling `ID3D10Device::RSSetViewports`.

```
D3D10_VIEWPORT vp[1];
vp[1].Width = 640.0f;
vp[1].Height = 480.0f;
vp[1].MinDepth = 0;
vp[1].MaxDepth = 1;
```

```
vp[1].TopLeftX = 0;  
vp[1].TopLeftY = 0;  
g_pd3dDevice->RSSetViewports( 1, vp );
```

The viewport description specifies the size of the viewport, the range to map depth to (using MinDepth and MaxDepth), and the placement of the top left of the viewport. MinDepth must be less than or equal to MaxDepth; the range for both MinDepth and MaxDepth is between 0.0 and 1.0, inclusive. It is common for the viewport to map to a render target but it is not necessary; additionally, the viewport does not have to have the same size or position as the render target.

You can create an array of viewports, but only one can be applied to a primitive output from the geometry shader. Only one viewport can be set active at a time. The pipeline uses a default viewport (and scissor rectangle, discussed in the next section) during rasterization. The default is always the first viewport (or scissor rectangle) in the array. To perform a per-primitive selection of the viewport in the geometry shader, specify the ViewportArrayIndex semantic on the appropriate GS output component in the GS output signature declaration.

The maximum number of viewports (and scissorrects) that can be bound to the rasterizer stage at any one time is 16 (specified with #define by

D3D10\_VIEWPORT\_AND\_SCISSORRECT\_OBJECT\_COUNT\_PER\_PIPELINE).

## Set the Scissor Rectangle

A scissor rectangle gives you another opportunity to reduce the number of pixels that will be sent to the output merger stage. Pixels outside of the scissor rectangle are discarded. Only one scissor rectangle at a time can be set to active in the geometry shader. The size of the scissor rectangle is specified in integers.

To enable the scissor rectangle, use the ScissorEnable member (in D3D10\_RASTERIZER\_DESC). The default scissor rectangle is an empty rectangle; that is, all rect values are 0. In other words, if you do not set up the scissor rectangle and scissor is enabled, you will not send any pixels to the output-merger stage. The most common setup is to initialize the scissor rectangle to the size of the viewport.

To set an array of scissor rectangles to the device, call ID3D10Device::RSSetScissorRects with D3D10\_RECT.xml.

```
D3D10_RECT rect[1];  
rect[0].left = 0;  
rect[0].right = 640;  
rect[0].top = 0;  
rect[0].bottom = 480;  
  
D3DDevice->RSSetScissorRects( 1, rect );
```

This method takes two parameters: (1) the number of rectangles in the array and (2) an array of rectangles.

The pipeline uses a default scissor rectangle index during rasterization (the default is a zero-size rectangle with clipping disabled). To override this, specify the SV\_ViewportArrayIndex semantic to a GS output component in the GS output signature declaration. This will cause the GS stage to mark this GS output component as a system-generated component with this semantic. The rasterizer stage recognizes this semantic and will use the parameter it is attached to as the scissor rectangle index to access the array of scissor rectangles. Don't forget to tell the rasterizer stage to use the scissor rectangle that you define by enabling the ScissorEnable value in the rasterizer description before creating the rasterizer object.

## Set Rasterizer State

In Direct3D 10, rasterizer state is encapsulated in a rasterizer state object. You may create up to 4096 rasterizer state objects which can then be set to the device by passing a handle to the state object.

Use ID3D10Device::CreateRasterizerState to create a rasterizer state object.

```
ID3D10RasterizerState * g_pRasterState;

D3D10_RASTERIZER_DESC rasterizerState;
rasterizerState.FillMode = D3D10_FILL_SOLID;
rasterizerState.CullMode = D3D10_CULL_FRONT;
rasterizerState.FrontCounterClockwise = true;
rasterizerState.DepthBias = false;
rasterizerState.DepthBiasClamp = 0;
rasterizerState.SlopeScaledDepthBias = 0;
rasterizerState.DepthClipEnable = true;
rasterizerState.ScissorEnable = true;
rasterizerState.MultisampleEnable = false;
rasterizerState.AntialiasedLineEnable = false;
pd3dDevice->CreateRasterizerState( &rasterizerState, &g_pRasterState );
```

This example set of state accomplishes perhaps the most basic rasterizer setup:

- Solid fill mode
- Cull out — that is, remove back faces; assume counter-clockwise winding order for primitives
- Turn off depth bias but enable depth buffering and enable the scissor rectangle
- Turn off multisampling and line antialiasing

In addition, basic rasterizer operations, always include the following: clipping (to the view frustum), perspective divide, and the viewport Scale. After successfully creating the rasterizer state object, set it to the device like this:

```
pd3dDevice->RSSetState(g_pRasterState);
```

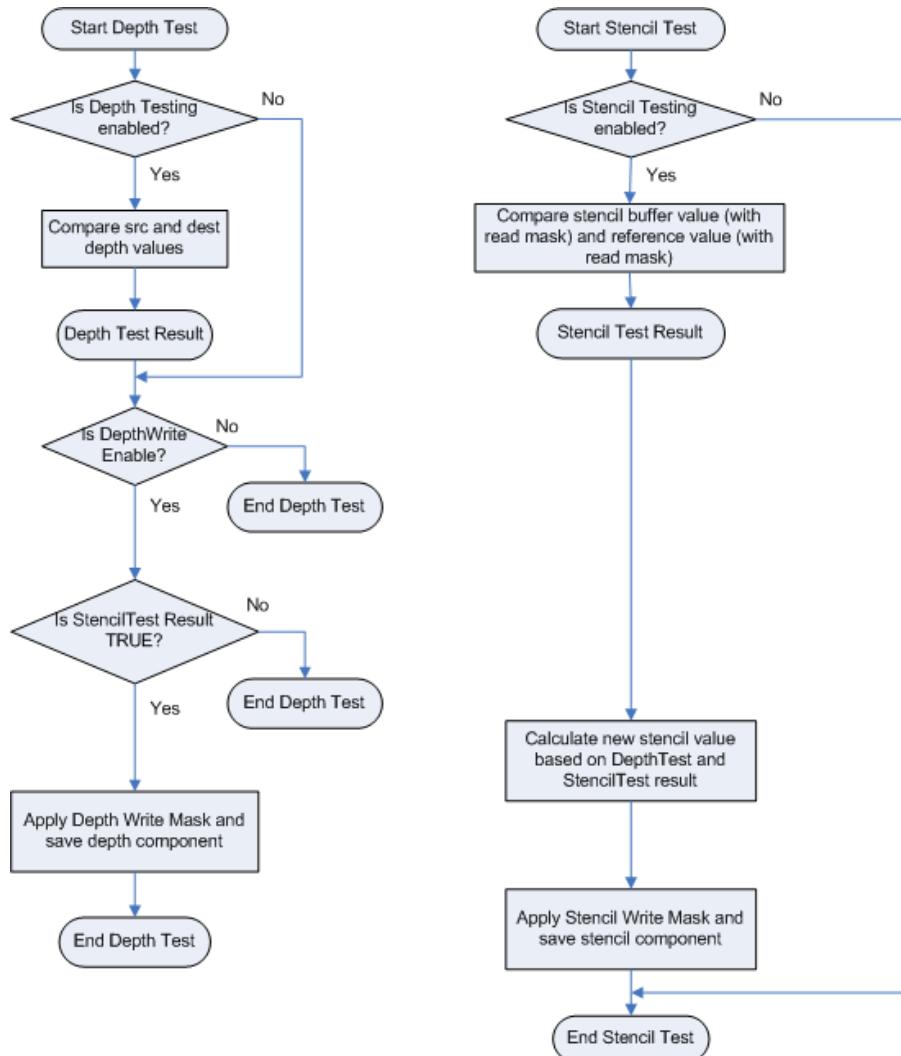
# Output-Merger Stage (Direct3D 10)

The output-merger (OM) stage generates the final rendered pixel color using a combination of pipeline state, the pixel data generated by the pixel shaders, the contents of the render targets, and the contents of the depth/stencil buffers. The OM stage is the final step for determining which pixels are visible (with depth-stencil testing) and blending the final pixel colors.

## Depth-Stencil Testing

A depth-stencil buffer, which is created as a 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. Ultimately, both the depth and stencil values data are used by the output-merger stage to determine if a pixel should be drawn or not. This flow chart shows conceptually how depth-stencil testing is done.

Figure 1. Depth-Stencil Testing Block Diagram



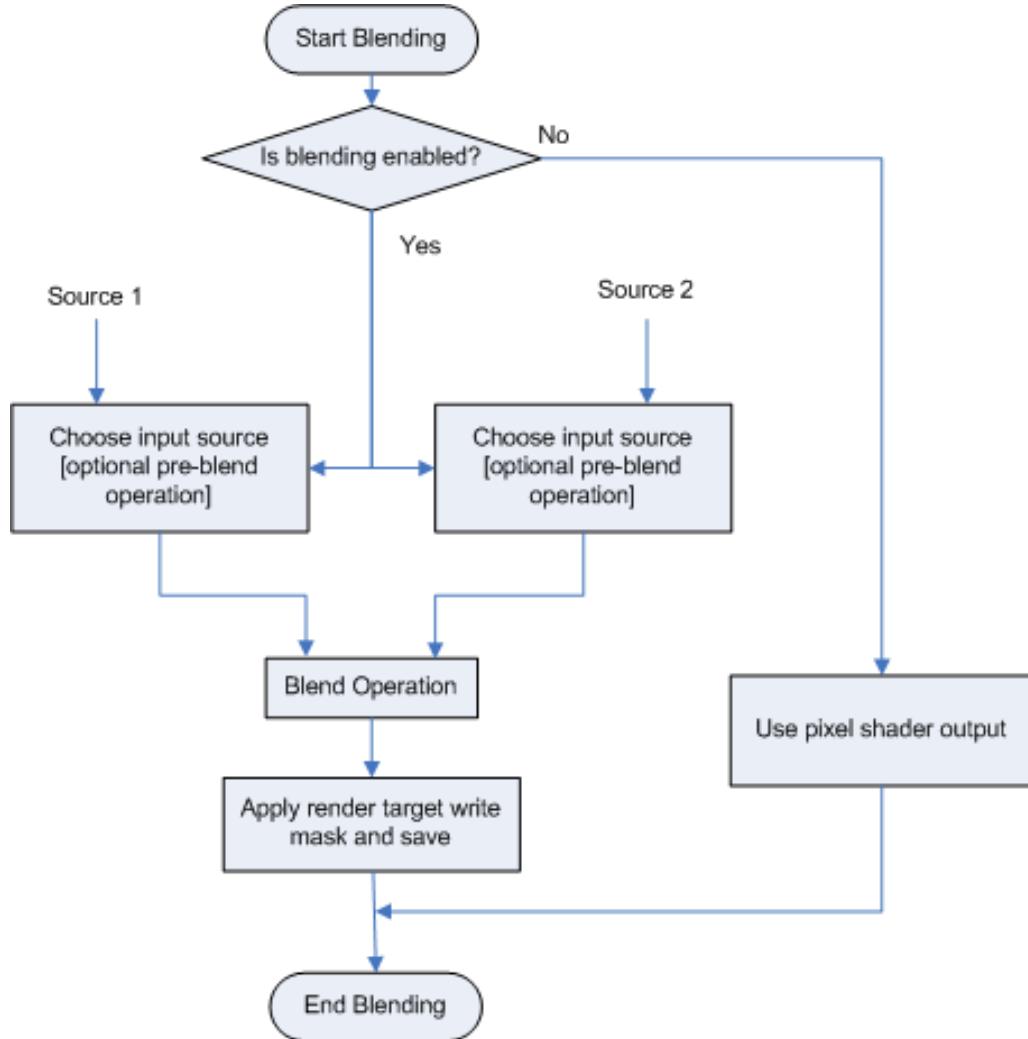
To configure depth-stencil testing, see [Configuring Depth-Stencil Functionality \(Direct3D 10\)](#). A depth-stencil object encapsulates depth-stencil state. An application can specify depth-stencil state, or the OM stage will use default values. Blending operations are performed on a per-pixel basis if multisampling is disabled. If multisampling is enabled, blending occurs on a per-multisample basis.

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

## Blending

Blending combines one or more pixel values to create a final pixel color. The following conceptual diagram describes the process involved in blending pixel data.

Figure 2. Blending Block Diagram



Conceptually, you can visualize this flow chart implemented twice in the output-merger stage: the first one blends RGB data, while in parallel, a second one blends alpha data. To see how to use the API to create and set blend state, see [Configuring Blending Functionality \(Direct3D 10\)](#).

# Configuring Depth-Stencil Functionality (Direct3D 10)

This section covers the steps for setting up the depth-stencil buffer, and depth-stencil state for the output-merger stage.

## Create a Depth-Stencil Resource

Create the depth-stencil buffer using a texture resource.

```
ID3D10Texture2D* pDepthStencil = NULL;
D3D10_TEXTURE2D_DESC descDepth;
descDepth.Width = backBufferSurfaceDesc.Width;
descDepth.Height = backBufferSurfaceDesc.Height;
descDepth.MipLevels = 1;
descDepth.ArraySize = 1;
descDepth.Format = pDeviceSettings->d3d10.AutoDepthStencilFormat;
descDepth.SampleDesc.Count = 1;
descDepth.SampleDesc.Quality = 0;
descDepth.Usage = D3D10_USAGE_DEFAULT;
descDepth.BindFlags = D3D10_BIND_DEPTH_STENCIL;
descDepth.CPUAccessFlags = 0;
descDepth.MiscFlags = 0;
hr = pd3dDevice->CreateTexture2D( &descDepth,           // Texture desc
                                    NULL,                  // Initial data
                                    &pDepthStencil ); // [out] Texture
```

## Create Depth-Stencil State

The depth-stencil state tells the output-merger stage how to perform the depth-stencil test. The depth-stencil test determines whether or not a given pixel should be drawn.

```
D3D10_DEPTH_STENCIL_DESC dsDesc;

// Depth test parameters
dsDesc.DepthEnable = true;
dsDesc.DepthWriteMask = D3D10_DEPTH_WRITE_MASK_ALL;
dsDesc.DepthFunc = D3D10_COMPARISON_LESS;

// Stencil test parameters
dsDesc.StencilEnable = true;
dsDesc.StencilReadMask = 0xFFFFFFFF;
dsDesc.StencilWriteMask = 0xFFFFFFFF;

// Stencil operations if pixel is front-facing
dsDesc.FrontFace.StencilFailOp = D3D10_STENCIL_OP_KEEP;
dsDesc.FrontFace.StencilDepthFailOp = D3D10_STENCIL_OP_INCR;
```

```

dsDesc.FrontFace.StencilPassOp = D3D10_STENCIL_OP_KEEP;
dsDesc.FrontFace.StencilFunc = D3D10_COMPARISON_ALWAYS;

// Stencil operations if pixel is back-facing
dsDesc.BackFace.StencilFailOp = D3D10_STENCIL_OP_KEEP;
dsDesc.BackFace.StencilDepthFailOp = D3D10_STENCIL_OP_DECR;
dsDesc.BackFace.StencilPassOp = D3D10_STENCIL_OP_KEEP;
dsDesc.BackFace.StencilFunc = D3D10_COMPARISON_ALWAYS;

// Create depth stencil state
ID3D10DepthStencilState * pDSState;
pDevice->CreateDepthStencilState(dsDesc, &pDSState);

```

## Bind Depth-Stencil Data to the OM Stage

Bind the depth-stencil state.

```
// Bind depth stencil state
pDevice->OMSetDepthStencilState(pDSState, 1);
```

Bind the depth-stencil resource using a view.

```

D3D10_DEPTH_STENCIL_VIEW_DESC descDSV;
descDSV.Format = DXGI_FORMAT_D32_FLOAT;
descDSV.ResourceType = D3D10_RESOURCE_TEXTURE2D;
descDSV.Texture2D.FirstArraySlice = 0;
descDSV.Texture2D.ArraySize = 1;
descDSV.Texture2D.MipSlice = 0;

// Create the depth stencil view
ID3D10DepthStencilView* pDSV;
hr = pd3dDevice->CreateDepthStencilView( pDepthStencil,
  // Depth stencil texture
  &descDSV, // Depth stencil desc
  &pDSV );
  // [out] Depth stencil view

// Bind the depth stencil view
pd3dDevice->OMSetRenderTargets( 1,      // One rendertarget view
                                  &pRTV, // Render target view, created earlier
                                  pDSV );
                                  // Depth stencil view for the render target

```

An array of rendertarget views may be passed into ID3D10Device::OMSetRenderTargets, however all of those rendertarget views will correspond to a single depth stencil view.

Rendertargets must all be the same type of resource. If multisample antialiasing is used, all bound rendertargets and depth buffers must have the same sample counts.

When a buffer is used as a rendertarget, depth-stencil testing and multiple rendertargets are not supported.

- As many as 8 render targets can be bound simultaneously.
- All rendertargets must have the same size in all dimensions (width and height, and depth for 3D or array size for \*Array types).
- Each rendertarget may have a different data format.
- Write masks control what data gets written to a rendertarget. The output write masks control on a per-rendertarget, per-component level what data gets written to the rendertarget(s).

## Advanced Stencil Techniques

The stencil portion of the depth-stencil buffer can be used for creating rendering effects such as compositing, decaling, and outlining.

### Compositing

Your application can use the stencil buffer to composite 2D or 3D images onto a 3D scene. A mask in the stencil buffer is used to occlude an area of the rendering target surface. Stored 2D information, such as text or bitmaps, can then be written to the occluded area. Alternately, your application can render additional 3D primitives to the stencil-masked region of the rendering target surface. It can even render an entire scene.

Games often composite multiple 3D scenes together. For instance, driving games typically display a rear-view mirror. The mirror contains the view of the 3D scene behind the driver. It is essentially a second 3D scene composited with the driver's forward view.

### Decaling

Direct3D applications use decaling to control which pixels from a particular primitive image are drawn to the rendering target surface. Applications apply decals to the images of primitives to enable coplanar polygons to render correctly.

For instance, when applying tire marks and yellow lines to a roadway, the markings should appear directly on top of the road. However, the z values of the markings and the road are the same. Therefore, the depth buffer might not produce a clean separation between the two. Some pixels in the back primitive may be rendered on top of the front primitive and vice versa. The resulting image appears to shimmer from frame to frame. This effect is called z-fighting or flimmering.

To solve this problem, use a stencil to mask the section of the back primitive where the decal will appear. Turn off z-buffering and render the image of the front primitive into the masked-off area of the render-target surface.

Multiple texture blending can be used to solve this problem.

## Outlines and Silhouettes

You can use the stencil buffer for more abstract effects, such as outlining and silhouetting.

If your application does two render passes - one to generate the stencil mask and second to apply the stencil mask to the image, but with the primitives slightly smaller on the second pass - the resulting image will contain only the primitive's outline. The application can then fill the stencil-masked area of the image with a solid color, giving the primitive an embossed look.

If the stencil mask is the same size and shape as the primitive you are rendering, the resulting image contains a hole where the primitive should be. Your application can then fill the hole with black to produce a silhouette of the primitive.

## Two-Sided Stencil

Shadow Volumes are used for drawing shadows with the stencil buffer. The application computes the shadow volumes cast by occluding geometry, by computing the silhouette edges and extruding them away from the light into a set of 3D volumes. These volumes are then rendered twice into the stencil buffer.

The first render draws forward-facing polygons, and increments the stencil-buffer values. The second render draws the back-facing polygons of the shadow volume, and decrements the stencil buffer values. Normally, all incremented and decremented values cancel each other out. However, the scene was already rendered with normal geometry causing some pixels to fail the z-buffer test as the shadow volume is rendered. Values left in the stencil buffer correspond to pixels that are in the shadow. These remaining stencil-buffer contents are used as a mask, to alpha-blend a large, all-encompassing black quad into the scene. With the stencil buffer acting as a mask, the result is to darken pixels that are in the shadows.

This means that the shadow geometry is drawn twice per light source, hence putting pressure on the vertex throughput of the GPU. The two-sided stencil feature has been designed to mitigate this situation. In this approach, there are two sets of stencil state (named below), one set each for the front-facing triangles and the other for the back-facing triangles. This way, only a single pass is drawn per shadow volume, per light.

An example of two-sided stencil implementation can be found in the `ShadowVolume10` Sample.

# Configuring Blending Functionality (Direct3D 10)

Blending operations are performed on every pixel shader output (RGBA value) before the output value is written to a render target. If multisampling is enabled, blending is done on each multisample; otherwise, blending is performed on each pixel.

## Create the Blend State

The blend state is a collection of states used to control blending. These states (defined in `D3D10_BLEND_DESC`) are used to create the blend state object by calling `ID3D10Device::CreateBlendState`.

For instance, here is a very simple example of blend-state creation that disables alpha blending and uses no per-component pixel masking.

```
ID3D10BlendState* g_pBlendStateNoBlend = NULL;

D3D10_BLEND_DESC BlendState;
ZeroMemory(&BlendState, sizeof(D3D10_BLEND_DESC));
BlendState.BlendEnable[0] = FALSE;
BlendState.RenderTargetWriteMask[0] = D3D10_COLOR_WRITE_ENABLE_ALL;
pd3dDevice->CreateBlendState(&BlendState, &g_pBlendStateNoBlend);
```

This example is from the [HLSLWithoutFX10 Sample](#).

## Bind the Blend State

Once the blend-state object is created, bind the blend-state object to the output-merger stage by calling `ID3D10Device::OMSetBlendState`.

```
float blendFactor = 0;
UINT sampleMask = 0xffffffff;

pd3dDevice->OMSetBlendState(g_pBlendStateNoBlend, blendFactor, sampleMask);
```

This API takes three parameters: the blend-state object, a four-component blend factor, and a sample mask. You may pass in `NULL` for the blend-state object (which tells the runtime to use default blend state - see `ID3D10Device::OMSetBlendState`) or pass in a blend-state object as shown here. The blend factor gives you per-component control over blending the new per-pixel values with the existing value. The sample mask is a user-defined mask that determines how to sample the existing rendertarget before updating it. The default sampling mask is `0xffffffff` which designates point sampling.

## Advanced Blending Topics

### Alpha-To-Coverage

Alpha-to-coverage is a multisampling technique that is most useful for situations such as dense foliage where there are several overlapping polygons. It can be turned on by setting the AlphaToCoverageEnable variable to true in the D3D10\_BLEND\_DESC. Alpha-to-coverage will work regardless of whether or not multisampling is also turned on in the rasterizer state (by setting MultisampleEnable to true or false).

Alpha-to-coverage works by taking the alpha component of an rgba value after it is output from a pixel shader and converting it into an n-step coverage mask (where n is the sample count). This n-step coverage mask is then ANDed with the multisample coverage mask and the result is used to determine which samples should get updated for all of the rendertargets currently bound to the output merger. The original alpha value in the output of the pixel shader is not changed when that alpha value is used to create the n-step coverage mask (alpha blending will still occur on a per-sample basis). Alpha-to-coverage multisampling is essentially the same as regular multisampling except that the n-step coverage mask is generated and ANDed with the multisample coverage mask.

For an example of alpha-to-coverage, see the Instancing10 Sample.

### Dual Source Color Blending

This feature enables the output merger to use both the pixel shader outputs simultaneously as input sources to a blending operation with the single rendertarget at slot 0.

Additional options are available for the SrcBlend, DestBlend, SrcBlendAlpha or DestBlendAlpha terms in the blend equation. The presence of any of the following choices in the blend equation means that dual source color blending is enabled:

```
D3D10_BLEND_SRC1COLOR  
D3D10_BLEND_INVSRC1COLOR  
D3D10_BLEND_SRC1ALPHA  
D3D10_BLEND_INVSRC1ALPHA
```

When dual source color blending is enabled, the pixel shader must have only a single rendertarget bound, at slot 0, and must output both SV\_TARGET0 and SV\_TARGET1. Writing SV\_Depth is still valid when performing dual source color blending.

The configured blend equation and the rendertarget write mask at slot 0 imply exactly which components from pixel shader outputs must be present. If the expected output components are not present, results are undefined. If extra components are output, they are ignored.

Examples:

There are times when a Shader computes 2 results that are useful on a single pass, but needs to combine one into the destination with a multiply and the other in with an add. This would look like:

```
SrcBlend = D3D10_BLEND_ONE;  
DestBlend = D3D10_BLEND_SRC1COLOR;
```

Next is a blend mode setup that takes pixel shader output color SV\_TARGET0 as source color, and uses pixel shader output color SV\_TARGET1 to blend with the destination color.

```
SrcBlend = D3D10_BLEND_SRC1COLOR;  
DestBlend = D3D10_BLEND_INVSRC1COLOR;
```

Example illustrating expected outputs from the pixel shader:

```
SrcBlend = D3D10_BLEND_SRC1ALPHA;  
DestBlend = D3D10_BLEND_SRCCOLOR;  
RenderTargetWriteMask[0] = ( D3D10_COLOR_WRITE_ENABLE_RED |  
                           D3D10_COLOR_WRITE_ENABLE_ALPHA );
```

# Resources (Direct3D 10)

A resource is an area in memory that can be accessed by the Direct3D pipeline. In order for the pipeline to access memory efficiently, data that is provided to the pipeline (such as input geometry, shader resources, textures etc) must be stored in a resource. There are two types of resources from which all Direct3D resources derive: a buffer or a texture.

Each application will typically create many resources. Examples of resource include: vertex buffers, index buffer, constant buffer, textures, and shader resources. There are several options that determine how resources can be used. You can create resources that are strongly typed or typeless; you can control whether resources have both read and write access; you can make resources accessible to only the CPU, GPU, or both. Naturally, there will be speed vs. functionality tradeoff - the more functionality you allow a resource to have, the less performance you should expect.

Since an application often uses many textures, Direct3D also introduces the concept of a texture array to simplify texture management. A texture array contains one or more textures (all of the same type and dimensions) that can be indexed from within an application or by shaders. Texture arrays allow you to use a single interface with multiple indexes to access many textures. You can create as many texture arrays to manage different texture types as you need.

Once you have created the resources that your application will use, you connect or bind each resource to the pipeline stages that will use them. This is accomplished by calling a bind API, which takes a pointer to the resource. Since more than one pipeline stage may need access to the same resource, Direct3D 10 introduces the concept of a resource view. A view identifies the portion of a resource that can be accessed. You can create m views or a resource and bind them to n pipeline stages, assuming you follow binding rules for shared resource (the runtime will generate errors at compile time if you don't).

A resource view provides a general model for access to a resource (textures, buffers, etc.). Because you can use a view to tell the runtime what data to access and how to access it, resource views allow you create typeless resources. That is, you can create resources for a given size at compile time, and then declare the data type within the resource when the resource gets bound to the pipeline. Views expose many new capabilities for using resources, such as the ability to read back depth/stencil surfaces in the shader, generating dynamic cubemaps in a single pass, and rendering simultaneously to multiple slices of a volume.

# Resource Types (Direct3D 10)

All resources used by the Direct3D pipeline derive from two basic resource types: buffers and textures. A buffer is a collection of elements; a texture is a collection of texels.

There are two ways to fully specify the layout (or memory footprint) of a resource:

Typed - fully specify the type when the resource is created.

Typeless - fully specify the type when the resource is bound to the pipeline.

## Buffer Resource

A buffer resource is a collection of data; each piece of data is called an element. Each element can be of a different type, or even a different format. For example, an application could store both index and vertex information in the same buffer. When the buffer is bound to the pipeline, an application specifies the information necessary to read and interpret buffer data.

A buffer is created as an unstructured resource (in essence just a big chunk of memory) which is interpreted when it is bound to a pipeline stage. Unlike a texture resource, a buffer cannot be filtered, does not contain multiple subresources, and cannot be multisampled.

## Buffer Element

An element is the smallest unit of memory that can be read or written by the pipeline. An element can be read by a shader or set as state in a Direct3D device. Some examples of an element could be: a position, a vertex normal, or a texture coordinate in a vertex buffer, and index in an index buffer, or a single state, such as depth/stencil.

An element is made up of 1 to 4 components. Examples of an element include: a packed data value (like R8G8B8A8), a single 8-bit integer, four 32-bit float values, etc. Specifically, an element is any one of the DXGI formats.

Although this description of elements applies to both buffers and textures, the documentation will use the term element when referring to buffers, and texel (short for texture element) when referring to textures for the sake of clarity.

- Vertex Buffer
- Index Buffer
- Shader-Constant Buffer

## Vertex Buffer

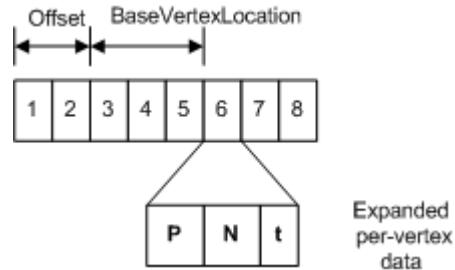
A vertex buffer is a collection of elements. A vertex buffer which contains position data can be visualized like this.

Figure 1. Single-Element Vertex Buffer



More often, a vertex buffer contains all the data needed to fully specify 3D vertices. This data is usually organized as structures of elements, which can be visualized like this.

Figure 2. Multi-Element Vertex Buffer



This vertex buffer contains eight vertices; the data for each vertex is made up of three elements (position, normal, and texture coordinates). The position and normal are typically specified using three 32-bit floats (DXGI\_FORMAT\_R32G32B32\_FLOAT) and the texture coordinates using two 32-bit floats (DXGI\_FORMAT\_R32G32\_FLOAT). Every element in a vertex buffer has an identical data structure to every other element.

To access data from a vertex buffer you need to know which vertex to access and these other buffer parameters:

Offset - the number of bytes from the start of the buffer to the data for the first vertex. The offset is supplied to ID3D10Device::IASetVertexBuffers.

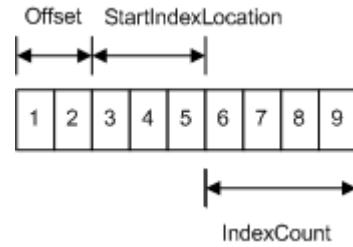
BaseVertexLocation - the number of bytes from the offset to the first vertex used by the appropriate draw call (see Draw APIs).

To create a buffer resource that can be used as a vertex buffer, see Create a Vertex Buffer.

## Index Buffer

An index buffer contains a sequential set of indices; each index is used to identify a vertex in a vertex buffer. Using an index buffer with one or more vertex buffers to supply data to the IA stage is called indexing. An index buffer can be visualized like this.

Figure 3. Index Buffer



The sequential indices stored in an index buffer are located with the following parameters:

- Offset - the number of bytes from the start of the buffer to the first index. The offset is supplied to `ID3D10Device::IASetIndexBuffer`.
- StartIndexLocation - the number of bytes from the offset to the first vertex used by the appropriate draw call (see Draw APIs).
- IndexCount - the number of indices to render.

An index buffer contains 16-bit or 32-bit indices. To create an index buffer, see [Create an Index Buffer](#).

### Shader-Constant Buffer

Direct3D 10 introduced a new buffer for supplying shader constants. It is called a shader-constant buffer. Conceptually, it looks just like a vertex buffer).

To create a shader-constant buffer, specify the constant-buffer bind flag (`D3D10_BIND_CONSTANT_BUFFER`). Once created, read the buffer using the `Load` (Direct3D 10 HLSL) intrinsic function. Each shader stage allows up to 15 slots for shader-constant buffers; each buffer can hold up to 4096 constants.

### Texture Resource

A texture resource is a structured collection of data designed to store texture data. The data in a texture resource is made up of one or more subresources, which are themselves organized into arrays of texels. Unlike buffers, textures can be filtered by texture samplers as they are read by shader units. The type of texture impacts how the texture is filtered.

- Texels
- Texture1D and Texture1DArray Resource
- Texture2D and Texture2DArray Resource
- Texture3D Resource

## Texels

A texel (or texture element) represents the smallest unit of a texture that can be read or written to by the pipeline. Each texel contains 1 to 4 components, arranged in one of the DXGI formats.

Individual texel components cannot be fetched (read) directly. The entire texel must be fetched by the runtime before an application can access a specific component. For example, a shader cannot fetch just the green component of an R8G8B8A8 texture; it would need to fetch the entire texel first and then access the green component.

## Texture1D and Texture1DArray Resource

A Texture1D resource is a one-dimension texture which consists of an array of elements. Unlike buffer resources, texture1D resources must be created with a texel type, may be filtered, and may contain one or more mipmap levels. Conceptually, a texture1D resource looks like this:

Figure 4. Texture1D Resource

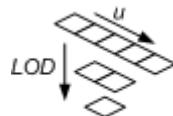
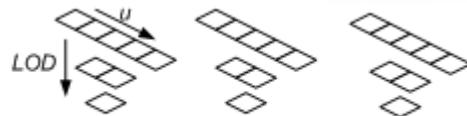


Figure 2 illustrates a Texture1D resource with 3 mipmap levels. The top-most mipmap level is the largest level; each successive level is a power of 2 (on each side) smaller. In this example, since the top-level texture width is 5 elements, there are two mipmap levels before the texture width is reduced to 1. Each element in each mipmap level is addressable by the *u* vector (which is commonly called a texture coordinate).

Each element in each mipmap level contains a single texel, or texture value. The data type of each element is defined by the texel format which is once again a DXGI\_FORMAT value. A texture resource may be typed or typeless at resource-creation time, but when bound to the pipeline, its interpretation must be provided in a view.

A Texture1DArray resource is a homogenous array of 1D textures. It looks like an array of 1D textures.

Figure 5. Texture1DArray Resource



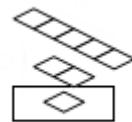
This texture array contains three textures. Each of the three textures has a texture width of 5 (which is the number of elements in the 1st layer). Each texture also contains a 3 layer mipmap.

All texture arrays in Direct3D are a homogenous array of textures; this means that every texture in a texture array must have the same data format and size (including texture width and number of mipmap levels). You may create texture arrays of different sizes, as long as all the textures in each array match in size.

### ***Subresources***

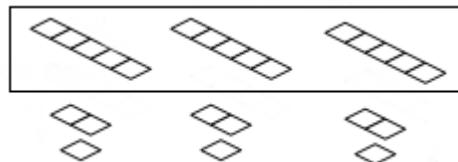
One interesting feature with Direct3D texture resources (including textures and texture arrays) is that they are made up of subresources. A subresource is a texture and mipmap-level combination. For a single texture, a subresource is a single mipmap level. This 1D texture is made up of 3 subresources.

Figure 6. Texture1D Subresource



For a texture array, a subresource is an array slice and a particular mipmap level. Here is an example of an array of subresources, within a 2D texture array.

Figure 7. Texture1DArray Subresource

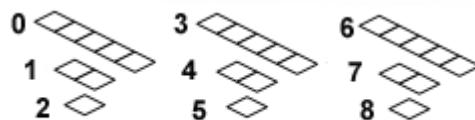


This array of subresources contains the top mipmap levels of all three textures in the texture array resource. Direct3D uses a resource view to access this array of texture subresources in a texture array.

### **Indexing Subresources**

A texture array can contain multiple textures, each with multiple mipmap levels. Each subresource is a mipmap-level in a single texture. When accessing a particular subresource, this is how the subresources are indexed within the texture array.

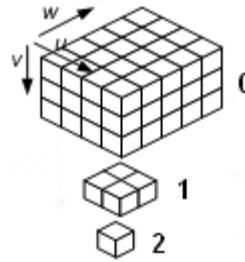
Figure 8. Indexing Subresources in a Texture1DArray



The index (which is an unsigned integer) starts at zero in the first texture in the array, and increments through the mipmap levels for that texture. Subresource indices in 2D texture arrays follow the same pattern as 1D texture arrays.

Subresources are indexed somewhat differently in a 3D texture resource (also known as a volume texture). Each subresource is a mipmap level in the Texture3D.

Figure 9. Indexing Subresources in a Texture3D



The index starts at zero in the first mipmap level in the Texture3D and increments through the mipmap levels for the rest of the texture.

#### Use D3D10CalcSubresource to Compute a Subresource Index

Many texture access methods require the application to specify a particular subresource. The helper function D3D10CalcSubresource (defined in D3D10.h) computes the subresource index from three pieces of information: the array slice, the mip slice, and the number of miplevels. The method is defined as follows:

```
UINT D3D10CalcSubresource( UINT MipSlice, UINT ArraySlice, UINT MipLevels )
{
    return MipSlice + ArraySlice * MipLevels;
}
```

#### Texture2D and Texture2DArray Resource

A Texture2D resource contains a 2D grid of texels. Each texel is addressable by a  $u, v$  vector. Since it is a texture resource, it may contain mipmap levels, and subresources. A fully populated 2D texture resource looks like this:

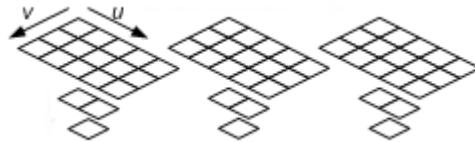
Figure 10. Texture2D Resource



This texture resource contains a single 3x5 texture with three mipmap levels.

A Texture2DArray resource is a homogeneous array of 2D textures; that is, each texture has the same data format and dimensions (including mipmap levels). It has a similar layout as the Texture1DArray resource except that the textures now contain 2D data, and therefore looks like this:

Figure 11. Texture2DArray Resource

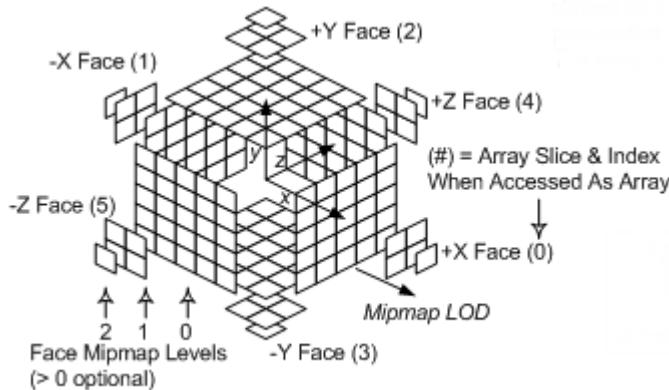


This texture array contains three textures; each texture is 3x5 with two mipmap levels.

### Using a Texture2DArray as a Texture Cube

A texture cube is a Texture2DArray that contains 6 textures, one for each face of the cube. A fully populated texture cube looks like this:

Figure 12. Array of 2D Textures Representing a Texture Cube

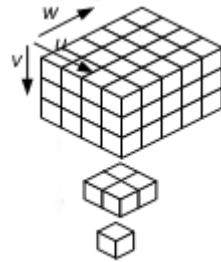


A Texture2DArray that contains 6 textures may be read from within shaders with the cube map intrinsic functions, after they are bound to the pipeline with a cube-texture view. Texture cubes are addressed from the shader with a 3D vector pointing out from the center of the texture cube.

### Texture3D Resource

A Texture3D resource (also known as a volume texture) contains a 3D volume of texels. Since it is a texture resource, it may contain mipmap levels. A fully populated Texture3D resource looks like this:

Figure 13. Texture3D Resource



When a Texture3D mipmap slice is bound as a rendertarget output, (by creating a rendertarget view), the Texture3D behaves identically to a Texture2DArray with n array slices where n is the depth (3rd dimension) of the Texture3D. The particular slice in the Texture3D to render is chosen from the geometry shader stage, by declaring a scalar component of output data as the `SV_RenderTargetArrayIndex` system-value.

There is no concept of an array of Texture3D resources; therefore a Texture3D subresource is a single mipmap level.

# Choosing a Resource (Direct3D 10)

A resource is a collection of data that is used by the 3D pipeline. Creating resources and defining their behavior is the first step toward programming your application. This guide covers basic topics for choosing the resources required by your application.

## Identify Pipeline Stages That Need Resources

The first step is to choose the pipeline stage (or stages) that will use a resource. That is, identify each stage that will read data from a resource as well as the stages that will write data out to a resource. Knowing the pipeline stages where the resources will be used determines the APIs that will be called to bind the resource to the stage.

This table lists the types of resources that can be bound to each pipeline stage. It includes whether the resource can be bound as an input or an output, as well as the bind API.

| Pipeline Stage  | In/Out | Resource               | Resource Type                                 | Bind API                                                                                                         |
|-----------------|--------|------------------------|-----------------------------------------------|------------------------------------------------------------------------------------------------------------------|
| Input Assembler | In     | Vertex Buffer          | Buffer                                        | ID3D10Device::IASetVertexBuffers                                                                                 |
| Input Assembler | In     | Index Buffer           | Buffer                                        | ID3D10Device::IASetIndexBuffer                                                                                   |
| Shader Stages   | In     | Shader-ResourceView    | Texture1D,<br>Texture2D,<br>Texture3D         | ID3D10Device::VSSetShaderResources,<br>ID3D10Device::GSSetShaderResources,<br>ID3D10Device::PSSetShaderResources |
| Shader Stages   | In     | Shader-Constant Buffer | Buffer                                        | ID3D10Device::VSSetConstantBuffers,<br>ID3D10Device::GSSetConstantBuffers,<br>ID3D10Device::PSSetConstantBuffers |
| Stream Output   | Out    | Buffer                 | Buffer                                        | ID3D10Device::SOSetTargets                                                                                       |
| Output Merger   | Out    | Rendertarget View      | Buffer, Texture1D,<br>Texture2D,<br>Texture3D | ID3D10Device::OMSetRenderTargets                                                                                 |
| Output Merger   | Out    | Depth/Stencil View     | Texture1D,<br>Texture2D                       | ID3D10Device::OMSetDepthStencilTargets                                                                           |

## Identify How Each Resource Will Be Used

Once you have chosen the pipeline stages that your application will use (and therefore the resources that each stage will require), the next step is to determine how each resource will be used, that is, whether a resource can be accessed by the CPU or the GPU.

The hardware that your application is running on will have a minimum of at least one CPU and one GPU. To pick a usage value, consider which type of processor needs to read or write to the resource from the following options (see D3D10\_USAGE).

| Resource Usage | Can be updated by | Frequency of Update |
|----------------|-------------------|---------------------|
| Default        | GPU               | infrequently        |
| Dynamic        | CPU               | frequently          |
| Staging        | GPU               | n/a                 |
| Immutable      | CPU               | n/a                 |

Default usage should be used for a resource that is expected to be updated by the CPU infrequently (less than once per frame). Ideally, the CPU would never write directly to a resource with default usage so as to avoid potential performance penalties.

Dynamic usage should be used for a resource that the CPU updates relatively frequently (once or more per frame). A typical scenario for a dynamic resource would be to create dynamic vertex and index buffers that would be filled at runtime with data about the geometry visible from the point of view of the user for each frame. These buffers would be used to render only the geometry visible to the user for that frame.

Staging usage should be used to copy data to and from other resources. A typical scenario would be to copy data in a resource with default usage (which the CPU cannot access) to a resource with staging usage (which the CPU can access).

Immutable resources should be used when the data in the resource will never change.

If you are unsure what usage to choose, start with the default usage as it is expected to be used most often.

## Binding Resources to Pipeline Stages

A resource can be bound to more than one pipeline stage at the same time as long as the restrictions specified when the resource was created (usage flags, bind flags, cpu access flags) are met. More

specifically, a resource can be bound as an input and an output simultaneously as long as reading and writing part of a resource cannot happen at the same time.

When binding a resource, think about how the GPU and the CPU will access the resource. Resources that are designed for a single purpose (do not use multiple usage, bind, and cpu access flags) will more than likely result in better performance.

For example, consider the case of a render target used as a texture multiple times. It may be faster to have two resources: a render target and a texture used as a shader resource. Each resource would use only one bind flag (D3D10\_BIND\_RENDER\_TARGET or D3D10\_BIND\_SHADER\_RESOURCE). The data would be copied from the render-target texture to the shader texture using `ID3D10Device::CopyResource` or `ID3D10Device::CopySubresourceRegion`. This may improve performance by isolating the render-target write from the shader-texture read. Of course, the only way to be sure is to implement both approaches and measure the performance difference in your particular application.

# Creating Buffer Resources (Direct3D 10)

A resource is a collection of data that is used by the 3D pipeline. The size of a resource is limited by D3D10\_REQ\_RESOURCE\_SIZE\_IN\_MEGABYTES. Creating resources and defining their behavior is the first step towards the rendering of geometric data.

## Creating Vertex Buffers

The steps to creating a vertex buffer are as follows.

Initialize a buffer description to describe the data to be stored in the buffer.

Initialize a subresource description to fill the buffer with data.

Create the buffer using both of these descriptions.

### A Buffer Description

When creating a vertex buffer, a buffer description (see D3D10\_BUFFER\_DESC) is used to define how data is organized within the buffer, how the pipeline can access the buffer, and how the buffer will be used.

The following example demonstrates how to create a buffer description for a single triangle with vertices that contain position and color values.

```
struct SimpleVertex
{
    D3DXVECTOR3 Position;
    D3DXVECTOR3 Color;
};

D3D10_BUFFER_DESC bufferDesc;
bufferDesc.Usage          = D3D10_USAGE_DEFAULT;
bufferDesc.ByteWidth      = sizeof( SimpleVertex ) * 3;
bufferDesc.BindFlags      = D3D10_BIND_VERTEX_BUFFER;
bufferDesc.CPUAccessFlags = 0;
bufferDesc.MiscFlags      = 0;
```

In this example, the buffer description is initialized with almost all default settings for usage, CPU access and miscellaneous flags. The other settings are for the bind flag which identifies the resource as a vertex buffer only, and the size of the buffer.

The usage and CPU access flags are important for performance. These two flags together determine how often a resource gets accessed, what type of memory the resource could be loaded into, and what processor needs to access the resource. Default usage this resource will not be updated very often. Setting CPU access to 0 means that the CPU will not need to either read or write the resource. Taken in

combination, this means that the runtime can load the resource into the highest performing memory for the GPU since the resource does not require CPU access.

As expected, there is a tradeoff between best performance and any-time accessibility by either processor. For example, the default usage with no CPU access means that the resource can be made available to the GPU exclusively. This could include loading the resource into memory not directly accessible by the CPU. The resource could only be modified with `ID3D10Device::UpdateSubresource`.

### A Subresource Description

A resource is made up of subresources. Applications can optionally provide some initial data when the buffer is created to both create it and initialize it at the same time. This is accomplished by using `D3D10_SUBRESOURCE_DATA`. The subresource description points to the actual resource data, and also contains information about the size and layout of that data.

Resources created with the `D3D10_USAGE_IMMUTABLE` flag (see `D3D10_USAGE`) must be initialized at creation time; resources that use any of the other creation flags can be updated after initialization. This can be accomplished using `ID3D10Device::CopyResource`, `ID3D10Device::CopySubresourceRegion`, `ID3D10Device::UpdateSubresource`, or by accessing its underlying memory using the resource's `Map` method.

A buffer is just a collection of elements and is laid out as a 1D array. As a result, the system memory pitch and system memory slice pitch are both the same; the size of the vertex data declaration.

This description gives the pipeline a clear idea of how to access the elements in the resource.

### Create a Vertex Buffer

To create a vertex buffer, call `ID3D10Device::CreateBuffer` with the buffer description and a subresource description.

The following code snippet demonstrates how to create a vertex buffer from an array of vertex data declared by the application. First, a buffer description is created and filled with information about the data to be stored in the vertex buffer and how that data will be used. Next, a subresource description is created and filled with the actual information the vertex buffer will be initialized with. Finally, the vertex buffer is created by calling `ID3D10Device::CreateBuffer`

```
struct SimpleVertexCombined
{
    D3DXVECTOR3 Pos;
    D3DXVECTOR3 Col;
};

ID3D10InputLayout* g_pVertexLayout = NULL;
ID3D10Buffer*      g_pVertexBuffer[2] = { NULL, NULL };
ID3D10Buffer*      g_pIndexBuffer = NULL;
```

```

SimpleVertexCombined verticesCombo[] =
{
    D3DXVECTOR3( 0.0f, 0.5f, 0.5f ),
    D3DXVECTOR3( 0.0f, 0.0f, 0.5f ),
    D3DXVECTOR3( 0.5f, -0.5f, 0.5f ),
    D3DXVECTOR3( 0.5f, 0.0f, 0.0f ),
    D3DXVECTOR3( -0.5f, -0.5f, 0.5f ),
    D3DXVECTOR3( 0.0f, 0.5f, 0.0f ),
};

D3D10_BUFFER_DESC bufferDesc;
bufferDesc.Usage          = D3D10_USAGE_DEFAULT;
bbufferDescd.ByteWidth   = sizeof( SimpleVertexCombined ) * 3;
bufferDesc.BindFlags     = D3D10_BIND_VERTEX_BUFFER;
bufferDesc.CPUAccessFlags = 0;
bufferDesc.MiscFlags     = 0;

D3D10_SUBRESOURCE_DATA InitData;
InitData.pSysMem = verticesCombo;
InitData.SysMemPitch = 0;
InitData.SysMemSlicePitch = 0;
hr = g_pd3dDevice->CreateBuffer( &bd, &InitData, &g_pVertexBuffer[0] );

```

## Create an Index Buffer

To create an index buffer, call `ID3D10Device::CreateBuffer` with the buffer description and a subresource description. Creating an index buffer is very similar to creating a vertex buffer. The main difference is that the `BindFlags` member of the `D3D10_BUFFER_DESC` structure must be set to `D3D10_BIND_INDEX_BUFFER`.

The following code snippet demonstrates how to create an index buffer from an array of index data. First, a buffer description is created and filled with information about the data to be stored in the index buffer and how that data will be used. Next, a subresource description is created and filled with the actual information the index buffer will be initialized with.

```

ID3D10Buffer *g_pIndexBuffer = NULL;

// Create indices
unsigned int indices[] = { 0, 1, 2 };

D3D10_BUFFER_DESC bufferDesc;
bufferDesc.Usage          = D3D10_USAGE_DEFAULT;
bufferDesc.ByteWidth       = sizeof( unsigned int ) * 3;
bufferDesc.BindFlags      = D3D10_BIND_INDEX_BUFFER;
bufferDesc.CPUAccessFlags = 0;
bbufferDescd.MiscFlags   = 0;

D3D10_SUBRESOURCE_DATA InitData;
InitData.pSysMem = indices;

```

```
InitData.SysMemPitch = 0;
InitData.SysMemSlicePitch = 0;
hr = g_pd3dDevice->CreateBuffer( &bd, &InitData, &g_pIndexBuffer );
if( FAILED( hr ) )
    return hr;

g_pd3dDevice->IASetIndexBuffer( g_pIndexBuffer, DXGI_FORMAT_R32_UINT, 0 );
```

# Creating Texture Resources (Direct3D 10)

A texture resource is a structured collection of data. Typically, color values are stored in textures and accessed during rendering by the pipeline at various stages for both input and output. Creating textures and defining how they will be used is an important part of rendering interesting-looking scenes in Direct3D 10.

Even though textures typically contain color information, creating textures using different formats enables them to store different kinds of data. This data can then be leveraged by the Direct3D 10 pipeline in non-traditional ways.

All textures have limits on how much memory they consume, and how many texels they contain. These limits are specified by resource constants.

## Creating Textures from Files

In Direct3D 10, textures cannot be used directly by the pipeline. Instead, a shader-resource view must be created which refers to the particular texture. This shader-resource view can then be bound to the pipeline during rendering. D3DX provides a useful function (see [D3DX10CreateShaderResourceViewFromFile](#)) to both load a texture from a file and create the accompanying shader-resource view at the same time, as shown below.

```
ID3D10ShaderResourceView *pTextureRV = NULL;  
D3DXCreateShaderResourceViewFromFile( pDevice, L"myTexture.bmp", &pTextureRV  
);
```

The ID3D10ShaderResourceView interface returned by the [D3DX10CreateShaderResourceViewFromFile](#) function can later be used to retrieve the original ID3D10Resource interface if it is needed. This can be done by calling the [ID3D10View::GetResource](#) method. If you know the type of texture you're loading (1D, 2D, or 3D), then you can cast this pointer directly to the desired type.

On the other hand, a generic texture manager may implement something like the following to handle different the texture types.

```
ID3D10Resource pResource = NULL;  
pTextureRV->GetResource( &pResource );  
D3D10_RESOURCE_DIMENSION type;  
pResource->GetType( &type );  
  
switch( type )  
{  
    case D3D10_RESOURCE_DIMENSION_TEXTURE1D:  
    {  
        pTexture1D = (ID3D10Texture1D*)pResource;  
        // ...
```

```

        break;
    }

    case D3D10_RESOURCE_DIMENSION_TEXTURE2D:
    {
        pTexture2D = (ID3D10Texture2D*)pResource;
        // ...
        break;
    }

    case D3D10_RESOURCE_DIMENSION_TEXTURE3D:
    {
        pTexture3D = (ID3D10Texture3D*)pResource;
        break;
    }

    default:
    {
        // error
        break;
    }
}

```

## Creating Empty Textures

Sometimes applications will want to create a texture and compute the data to be stored in the texture, or use the graphics pipeline to render to this texture and later use the results in other processing. These textures could be updated by the graphics pipeline or by the application itself, depending on what kind of usage was specified for the texture when it was created.

### Rendering to a texture

The most common case of creating an empty texture to be filled with data during runtime is the case where an application wants to render to a texture and then use the results of the rendering operation in a subsequent pass. Textures created with this purpose should specify default usage.

The following code sample creates an empty texture that the pipeline can render to and subsequently use as an input to shader.

```

// Create the render target texture
D3D10_TEXTURE2D_DESC desc;
ZeroMemory( &desc, sizeof(desc) );
desc.Width = 256;
desc.Height = 256;
desc.MipLevels = 1;
desc.ArraySize = 1;
desc.Format = DXGI_FORMAT_R32G32B32A32_FLOAT;
desc.SampleDesc.Count = 1;
desc.Usage = D3D10_USAGE_DEFAULT;
desc.BindFlags = D3D10_BIND_RENDER_TARGET | D3D10_BIND_SHADER_RESOURCE;

```

```
ID3D10Texture2D pRenderTarget = NULL;
pDevice->CreateTexture2D( &desc, NULL, &pRenderTarget );
```

Creating the texture requires the application to specify some information about the properties the texture will have. The width and height of the texture in texels is set to 256. For this render target, a single mipmap is all we need. Only one render target is required so the array size is set to 1. Each texel contains four 32-bit floating point values, which can be used to store very precise information (see DXGI\_FORMAT). One sample per pixel is all that will be needed. The usage is set to default because this allows for the most efficient placement of the render target in memory. Finally, the fact that the texture will be bound as a render target and a shader resource at different points in time is specified.

Textures cannot be bound for rendering to the pipeline directly. Because of this, a render-target view must first be created to describe to the pipeline how to access the render target texture. This can be seen in the following code sample.

```
D3D10_RENDER_TARGET_VIEW_DESC rtDesc;
rtDesc.Format = desc.Format;
rtDesc.ViewDimension = D3D10_RTV_DIMENSION_TEXTURE2D;
rtDesc.Texture2D.MipSlice = 0;

ID3D10RenderTargetView pRenderTargetView = NULL;
pDevice->CreateRenderTargetView( pRenderTarget, &rtDesc, &pRenderTargetView ) ;
```

The format of the render-target view is simply set to the format of the original texture. The information in the resource should be interpreted as a 2D texture, and we only want to use the first mipmap of the render target.

Similarly to how a render-target view must be created so that the render target can be bound for output to the pipeline, a shader-resource view must be created so that the render target can be bound to the pipeline as an input. The following code sample demonstrates this.

```
// Create the shader-resource view
D3D10_SHADER_RESOURCE_VIEW_DESC srDesc;
srDesc.Format = desc.Format;
srDesc.ViewDimension = D3D10_SRV_DIMENSION_TEXTURE2D;
srDesc.Texture2D.MostDetailedMip = 0;
srDesc.Texture2D.MipLevels = 1;

ID3D10ShaderResourceView pShaderResView = NULL;
pDevice->CreateShaderResourceView( pRenderTarget, &srDesc, &pShaderResView ) ;
```

The parameters of shader-resource view descriptions are very similar to those of render-target view descriptions and were chosen for the same reasons.

## Filling Textures Manually

Sometimes applications would like to compute values at runtime, put them into a texture manually and then have the graphics pipeline use this texture in later rendering operations. To do this, the application must create an empty texture in such a way to allow the CPU to access the underlying memory. This is done by creating a dynamic texture and gaining access to the underlying memory by calling a particular method. The following code sample demonstrates how to do this.

```
D3D10_TEXTURE2D_DESC desc;
desc.Width = 256;
desc.Height = 256;
desc.MipLevels = desc.ArraySize = 1;
desc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
desc.SampleDesc.Count = 1;
desc.Usage = D3D10_USAGE_DYNAMIC;
desc.BindFlags = D3D10_BIND_SHADER_RESOURCE;
desc.CPUAccessFlags = D3D10_CPU_ACCESS_WRITE;
ID3D10Texture2D* pTexture = NULL;
pd3dDevice->CreateTexture2D( &desc, NULL, &pTexture );
```

Note that the format is set to a 32 bits per pixel where each component is defined by 8 bits. The usage parameter is set to dynamic while the bind flags are set to specify the texture will be accessed by a shader. The rest of the texture description is similar to creating a render target.

Calling ID3D10Texture2D::Map enables the application to access the underlying memory of the texture. The pointer retrieved is then used to fill the texture with data. This can be seen in the following code sample.

```
D3D10_MAPPED_TEXTURE2D mappedTex;
pTexture->Map( D3D10CalcSubresource(0, 0, 1), D3D10_MAP_WRITE_DISCARD, 0,
&mappedTex );

UCHAR* pTexels = (UCHAR*)mappedTex.pData;
for( UINT row = 0; row < desc.Height; row++ )
{
    UINT rowStart = row * mappedTex.RowPitch;
    for( UINT col = 0; col < desc.Width; col++ )
    {
        UINT colStart = col * 4;
        pTexels[rowStart + colStart + 0] = 255; // Red
        pTexels[rowStart + colStart + 1] = 128; // Green
        pTexels[rowStart + colStart + 2] = 64; // Blue
        pTexels[rowStart + colStart + 3] = 32; // Alpha
    }
}

pTexture->Unmap( D3D10CalcSubresource(0, 0, 1) );
```

## Multiple Rendertargets

Up to eight rendertarget views can be bound to the pipeline at a time (with a call to `ID3D10Device::OMSetRenderTargets`). For each pixel (or each sample if multisampling is enabled), the blend process will be done on each rendertarget view bound to the output merger. Two of the blend state variables - `BlendEnable` and `RenderTargetWriteMask` - are arrays of eight, with each member of the arrays corresponding to each rendertarget view set to the output merger. When setting multiple rendertargets to the output merger, each rendertarget must be the same type of resource (i.e. buffer, `texture1D[array]`, `texture2D[array]`, `texture3D`) and must have the same size in all dimensions (width, height, depth for `texture3Ds`, and array size for texture arrays). If the rendertargets are multisampled textures, then they must all have the same number of samples per pixel.

There can only be one Depth/Stencil buffer active, regardless of how many `RenderTarget`s are active. Should resource views of `TextureArray(s)` be set as `RenderTarget(s)`, the resource view of Depth/Stencil (if bound) must also be the same dimensions and array size. Note that this does not mean that the Resources, themselves, need to be of the same dimensions (including array size). Only that the views that are used together must be of the same effective dimensions.

The rendertargets need not be the same texture format.

# Copying and Accessing Resource Data (Direct3D 10)

It is no longer necessary to think about resources as being created in either video memory or system memory. Or whether or not the runtime should manage the memory. Thanks to the architecture of the new WDDM (Windows Display Driver Model), applications now create Direct3D 10 resources with different usage flags to indicate how the application intends on using the resource data. The new driver model virtualizes the memory used by resources; it then becomes the responsibility of the operating system/driver/memory manager to place resources in the most performant area of memory possible given the expected usage.

The default case is for resources to be available to the GPU. Of course, having said that, there are times when the resource data needs to be available to the CPU. Copying resource data around so that the appropriate processor can access it without impacting performance requires some knowledge of how the API methods work.

## Copying Resource Data

Resources are created in memory when Direct3D executes a Create call. They can be created in video memory, system memory, or any other kind of memory. Since WDDM driver model virtualizes this memory, applications no longer need to keep track of what kind of memory resources are created in.

Ideally, all resources would be located in video memory so that the GPU can have immediate access to them. However, it is sometimes necessary for the CPU to read the resource data or for the GPU to access resource data the CPU has written to. Direct3D 10 handles these different scenarios by requesting the application specify a usage type, and then offers several methods for copying resource data when necessary.

Depending on how the resource was created, it is not always possible to directly access the underlying data. This may mean that the resource data must be copied from the source resource to another resource that is accessible by the appropriate processor. In terms of Direct3D 10, default resources can be accessed directly by the GPU, dynamic and staging resources can be directly accessed by the CPU.

Once a resource has been created, its usage cannot be changed. Instead, copy the contents of one resource to another resource that was created with a different usage. Direct3D 10 provides this functionality with three different methods. The first two methods( `ID3D10Device::CopyResource` and `ID3D10Device::CopySubresourceRegion`) are designed to copy resource data from one resource to another. The third method (`ID3D10Device::UpdateSubresource`) is designed to copy data from memory to a resource.

There are two main kinds of resources: mappable and non-mappable. Resources created with dynamic or staging usages are mappable, while resources created with default or immutable usages are non-mappable.

Copying data among non-mappable resources is very fast because this is the most common case and has been optimized to perform well. Since these resources are not directly accessible by the CPU, they are optimized so that the GPU can manipulate them quickly.

Copying data among mappable resources is more problematic because the performance will depend on the usage the resource was created with. For example, the GPU can read a dynamic resource fairly quickly but cannot write to them, and the GPU cannot read or write to staging resources directly.

Applications that wish to copy data from a resource with default usage to a resource with staging usage (to allow the CPU to read the data -- i.e. the GPU readback problem) must do so with care. See [Accessing Resource Data](#) for more details on this last case.

## **Accessing Resource Data**

Accessing a resource requires mapping the resource; mapping essentially means the application is trying to give the CPU access to memory. The process of mapping a resource so that the CPU can access the underlying memory can cause some performance bottlenecks and for this reason, care must be taken as to how and when to perform this task.

Performance can grind to a halt if the application tries to map a resource at the wrong time. If the application tries to access the results of an operation before that operation is finished, a pipeline stall will occur.

Performing a map operation at the wrong time could potentially cause a severe drop in performance by forcing the GPU and the CPU to synchronize with each other. This synchronization will occur if the application wants to access a resource before the GPU is finished copying it into a resource the CPU can map.

The CPU can only read from resources created with the D3D10\_USAGE\_STAGING flag. Since resources created with this flag cannot be set as outputs of the pipeline, if the CPU wants to read the data in a resource generated by the GPU, the data must be copied to a resource created with the staging flag. The application may do this by using the ID3D10Device::CopyResource or ID3D10Device::CopySubresourceRegion methods to copy the contents of one resource to another. The application can then gain access to this resource by calling the appropriate Map method. When access to the resource is no longer needed, the application should then call the corresponding Unmap method. For example, ID3D10Texture2D::Map and ID3D10Texture2D::Unmap. The different Map methods return some specific values depending on the input flags. See [Map Remarks](#) section for details.

## **Performance Considerations**

It is best to think of a PC as a machine running as a parallel architecture with two main types of processors: one or more CPU's and one or more GPU's. As in any parallel architecture, the best performance is achieved when each processor is scheduled with enough tasks to prevent it from going idle and when the work of one processor is not waiting on the work of another.

The worst-case scenario for GPU/CPU parallelism is the need to force one processor to wait for the results of work done by another. Direct3D 10 tries to remove this cost by making the ID3D10Device::CopyResource and ID3D10Device::CopySubresourceRegion methods asynchronous; the copy has not necessarily executed by the time the method returns. The benefit of this is that the application does not pay the performance cost of actually copying the data until the CPU accesses the data, which is when Map is called. If the Map method is called after the data has actually been copied, no performance loss occurs. On the other hand, if the Map method is called before the data has been copied, then a pipeline stall will occur.

Asynchronous calls in Direct3D 10 (which are the vast majority of methods, and especially rendering calls) are stored in what is called a command buffer. This buffer is internal to the graphics driver and is used to batch calls to the underlying hardware so that the costly switch from user mode to kernel mode in Microsoft Windows occurs as rarely as possible.

The command buffer is flushed, thus causing a user/kernel mode switch, in one of four situations, which are as follows.

1. IDXGISwapChain::Present is called.
2. ID3D10Device::Flush is called.
3. The command buffer is full; its size is dynamic and is controlled by the Operating System and the graphics driver.
4. The CPU requires access to the results of a command waiting to execute in the command buffer.

Of the four situations above, number four is the most critical to performance. If the application issues a ID3D10Device::CopyResource or ID3D10Device::CopySubresourceRegion call, this call is queued in the command buffer. If the application then tries to map the staging resource that was the target of the copy call before the command buffer has been flushed, a pipeline stall will occur because not only does the Copy method call need to execute, but all other buffered commands in the command buffer must execute as well. This will cause the GPU and CPU to synchronize because the CPU will be waiting to access the staging resource while the GPU is emptying the command buffer and finally filling the resource the CPU needs. Once the GPU finishes the copy, the CPU will begin accessing the staging resource, but during this time, the GPU will be sitting idle.

Doing this frequently at runtime will severely degrade performance. For that reason, mapping of resources created with default usage should be done with care. The application needs to wait long enough for the command buffer to be emptied and thus have all of those commands finish executing before it tries to map the corresponding staging resource. How long should the application wait? At least two frames because this will enable parallelism between the CPU(s) and the GPU to be maximally leveraged. The way the GPU works is that while the application is processing frame N by submitting calls to the command buffer, the GPU is busy executing the calls from the previous frame, N-1.

So if an application wants to map a resource that originates in video memory and calls `ID3D10Device::CopyResource`/`ID3D10Device::CopySubresourceRegion` at frame N, this call will actually begin to execute at frame N+1, when the application is submitting calls for the next frame. The copy should be finished when the application is processing frame N+2.

The following table summarizes this discussion.

| Frame | GPU/CPU Status                                                                                                                                                                   |
|-------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| N     | CPU issues render calls for current frame.                                                                                                                                       |
| N+1   | GPU executing calls sent from CPU during frame N.<br>CPU issues render calls for current frame.                                                                                  |
| N+2   | GPU finished executing calls sent from CPU during frame N. Results ready.<br>GPU executing calls sent from CPU during frame N+1.<br>CPU issues render calls for current frame.   |
| N+3   | GPU finished executing calls sent from CPU during frame N+1. Results ready.<br>GPU executing calls sent from CPU during frame N+2.<br>CPU issues render calls for current frame. |
| N+4   | ...                                                                                                                                                                              |

# Memory Structure and Views (Direct3D 10)

Direct3D 10 enables memory to be allocated and accessed in very flexible ways. How the runtime accesses resources is determined in large part by how the memory is allocated.

## Memory Structure

Resources can be created with varying types of memory structures. Less restrictive memory structures offer applications more flexibility in how they use the data stored in resources, but may not be optimal for performance.

## Unstructured Resources

Unstructured resources are essentially resources allocated with a single contiguous block of memory. This memory is allocated without any knowledge of the type of data that will be stored in it. Only buffers can be allocated as unstructured resources.

## Structured Resources

Textures are created as a structured resource. These are split into two groups.

- Typeless
- Typed

### Typeless

In a structured, yet typeless resource, the data types of individual texels are not supplied when the resource is first created. The application must choose from the available typeless formats. This allows the runtime to allocate the appropriate amount of memory and generate a full mipmap chain. However, the exact data format (whether the memory will be interpreted as integers, floating point values, unsigned integers etc.) is not determined until the resource is bound to the pipeline with a view.

The format of each element in a typeless resource is specified when the resource is bound to a pipeline stage using a view. Because the resource is weakly typed storage, limited reuse or reinterpretation of the memory is enabled, as long as the component bit counts remain the same.

The same resource may be bound to multiple slots in the graphics pipeline with a view of different, fully qualified formats at each location. For example, a resource created with the format `DXGI_FORMAT_R32G32B32A32_TYPELESS` could be used as an `DXGI_FORMAT_R32G32B32A32_FLOAT` and an `DXGI_FORMAT_R32G32B32A32_UINT` at different locations in the pipeline simultaneously. Please see binding resources for more details.

### Typed

Creating a fully typed resource restricts the resource to the format it was created with. This enables the runtime to optimize access, especially if the resource is created with flags indicating that it cannot be mapped by the application.

Since the resource is created with a specific type, any views used with this type of resource must have identical formats as the resource itself. Resources created with a specific type cannot be reinterpreted using the view mechanism.

## Views

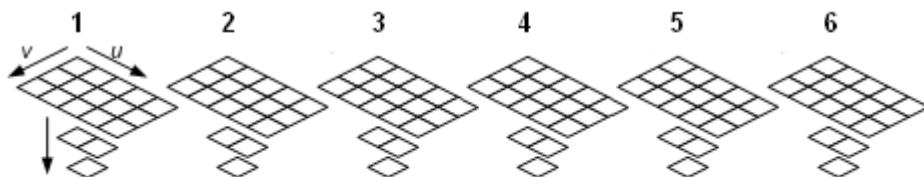
In Direct3D 10, texture resources are accessed with a view, which is a mechanism for hardware interpretation of a resource in memory. A view allows a particular pipeline stage to access only the subresources it needs, in the representation desired by the application.

A view supports the notion of a typeless resource - that is, the resource is created is of a certain size, but exactly how the memory is interpreted is determined when the resource is bound to the pipeline. See typeless resources for more details.

Here is an example of binding a Texture2DArray resource with 6 textures two different ways through two different views. (Note: a subresource cannot be bound as both input and output to the pipeline simultaneously.)

The Texture2DArray can be used as a shader resource by creating a shader resource view for it. The resource is then addressed as an array of textures.

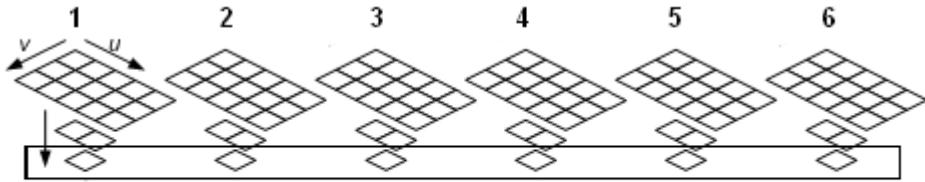
Figure 1. Texture2DArray Viewed as 6 2D Textures



Using a Texture2DArray as a render target. The resource can be viewed as an array of 2D textures (6 in this case) with mipmap levels (3 in this case).

Create a view object for a rendertarget by calling CreateRenderTargetView. Then call OMSetRenderTargets to set the rendertarget view to the pipeline. Render into the rendertargets by calling Draw and using the RenderTargetArrayIndex to index into the proper texture in the array. You can use a subresource (a mipmap level, array index combination) to bind to any array of subresources. So you could bind to the second mipmap level and only update this particular mipmap level if you wanted like this:

Figure 2. Views can access an array of Subresources



### Render-Target Views

Creating a rendertarget view for a resource will enable the user to bind the resource to the pipeline as a rendertarget. A rendertarget is a resource that will be written to by the output merger stage at the end of a render pass; after a render pass the rendertargets contain the color information of a rendered image. When creating a rendertarget view, only one mipmap level of a texture resource may be used as a rendertarget. When binding a rendertarget view to the pipeline, there will be a corresponding depth stencil view that will be used with the rendertarget in the output merger stage.

### Depth Stencil Views

Creating a depth stencil view for a resource will enable the user to use the resource as depth stencil. A depth stencil view is bound to the output merger stage along with a corresponding rendertarget view.

### Shader Resource Views

A shader resource view enables the user to bind a resource to a shader stage. A shader resource view can interpret a resource as something different from when it was created. For example, a shader resource view may interpret a Texture2DArray with six textures as a cube map. By binding this cube map view to a shader stage, the shader will sample the Texture2DArray as a cube map.

Differences between Direct3D 9 and Direct3D 10:

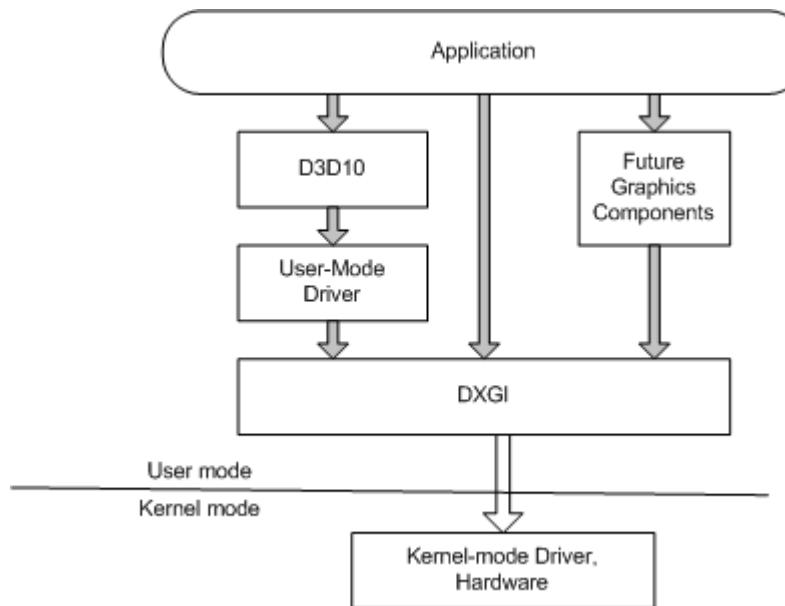
In Direct3D 10, you no longer bind a resource directly to the pipeline, you create a view of a resource, and then set the view to the pipeline. This allows validation and mapping in the runtime and driver to occur at view creation, minimizing type checking at bind-time.

# DXGI Overview (Direct3D 10)

DirectX Graphics Infrastructure (DXGI) recognizes that some parts of graphics evolve more slowly than others. The primary goal of DXGI is to manage low-level tasks that can be independent of the DirectX graphics runtime. DXGI provides a common framework for future graphics components, the first component that takes advantage of DXGI is Direct3D 10.

In previous versions of Direct3D, low-level tasks like enumeration of hardware devices, presenting rendered frames to an output, controlling gamma, and managing a full-screen transition were included in the 3D runtime. These tasks are now implemented in DXGI.

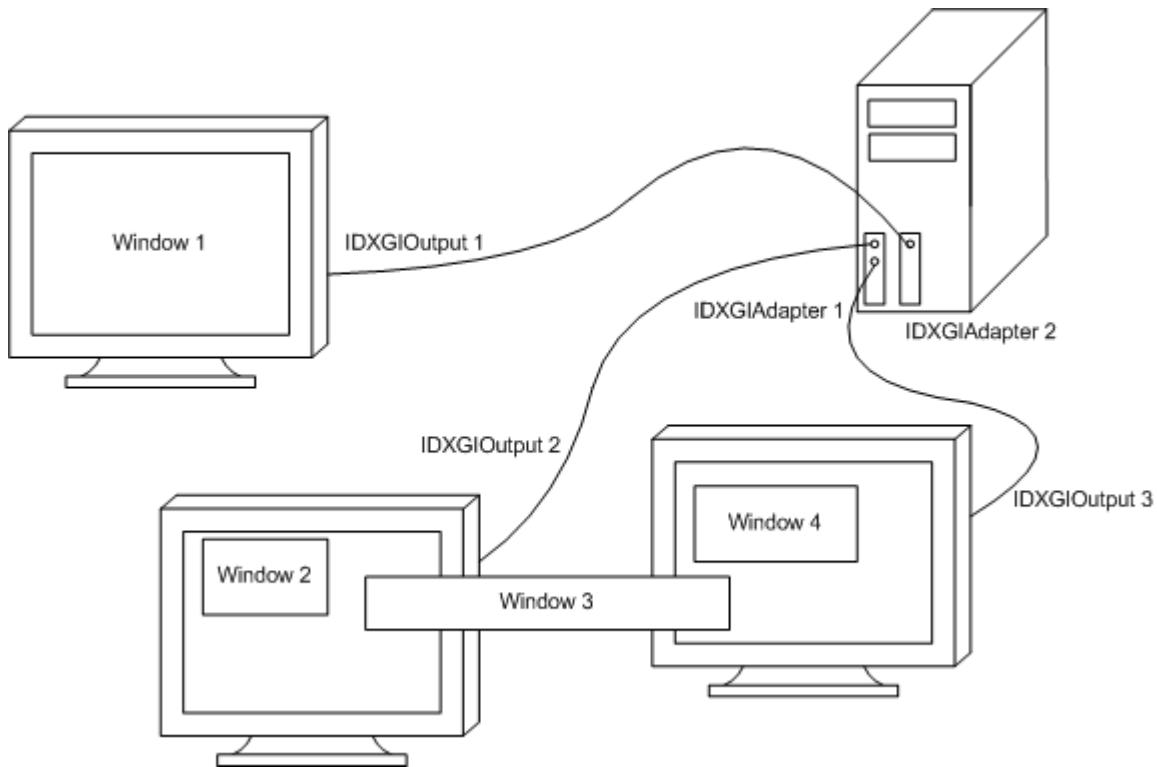
DXGI's purpose is to communicate with the kernel mode driver and the system hardware.



An application has the option of talking to DXGI directly, or calling the Direct3D APIs in D3D10Core (which handles the communications with DXGI for you). You may want to work with DXGI directly if your application needs to enumerate devices or control how data is presented to an output.

## Enumerating Adapters

An adapter is an abstraction of the hardware and the software capability of your computer. There are generally many adapters on your machine. Some devices are implemented in hardware (like your video card) and some are implemented in software (like the Direct3D reference rasterizer). Adapters implement functionality used by a graphic application. The following diagram shows a system with a single computer, two adapters (video cards) and three output monitors.



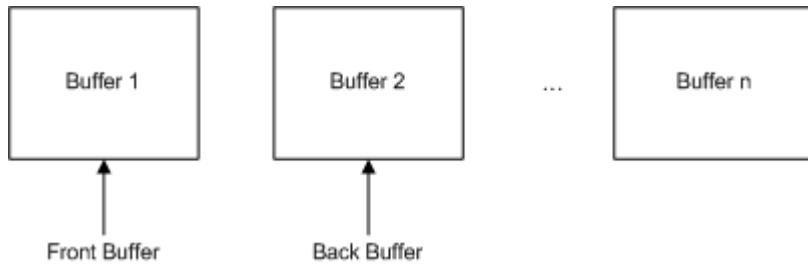
When enumerating these pieces of hardware, DXGI creates an IDXGIOOutput interface for each output (or monitor) and an IDXGIAdapter interface for each video card (even if it is a video card built into a motherboard). Enumeration is done by using an IDXGIFactory interface in two stages. First, call `IDXGIFactory::EnumAdapters` to return a set of IDXGIAdapter interfaces that represent the video hardware. Second, call `IDXGIAdapter::CheckInterfaceSupport` to see if the video hardware is associated with a driver that supports the API you wish to use (such as Direct3D 10).

If your application is calling Direct3D, there is no need to enumerate devices since the DirectX runtime will create a default adapter interface when `D3D10CreateDevice` is called.

## Presentation

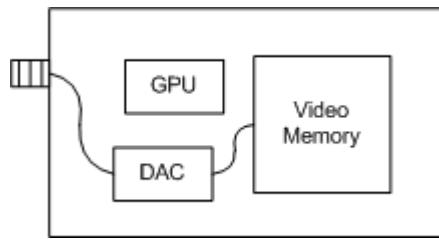
Your application's job is to render frames and ask DXGI to present those frames to the output. For performance, this requires at least two buffers so that the application can render one buffer while presenting another one. There may be more than two buffers required depending on the time it takes to render a frame or the desired frame rate for presentation. The set of buffers created is called a swap chain.

Figure 1. Swap Chain



A swap chain has one front buffer and one or more back buffers. Each application is responsible for creating its own swap chain. To maximize the speed of the presentation of the data to an output, a swap chain is almost always created in the memory in a display sub-system.

Figure 2. Display Sub-System



The display sub-system (which is often a video card but could be implemented on a motherboard) contains a GPU, a digital to analog converter (DAC) and memory. The swap chain is allocated within this memory to make presentation very fast. The display sub-system is responsible for presenting the data in the front buffer to the output.

### Create a Swap Chain

Differences between Direct3D 9 and Direct3D 10:

Direct3D 10 is the first graphics component to use DXGI. DXGI has some different swap chain behaviors.

- In DXGI, a swap chain is tied to a window when the swap chain is created. This change improves performance and saves memory. Previous versions of Direct3D allowed the swap chain to change the window that the swap chain is tied to.
- In DXGI, a swap chain is tied to a rendering device on creation. A change to the rendering device requires the swap chain to be recreated.

A swap chain's buffers are created at a particular size and in a particular format. The application specifies these values (or you can inherit the size from the target window) at startup, and can then optionally modify them as the window size changes in response to user input or program events.

Once the swapchain has been created, you will typically want to render images into it. Here's a code fragment that sets up a Direct3D10 context to render into a swapchain. This code extracts a buffer from the swapchain, creates a render-target-view from that buffer, then sets it on the device:

```
ID3D10Resource * pBB;
ThrowFailure( pSwapChain->GetBuffer(0, __uuidof(pBB),
    reinterpret_cast<void**>(&pBB)), "Couldn't get back buffer");
ID3D10RenderTargetView * pView;
D3D10_RENDER_TARGET_VIEW_DESC rtd;
rtd.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
rtd.ViewDimension = D3D10_RTV_DIMENSION_TEXTURE2D;
rtd.Texture2D.MipSlice = 0;
ThrowFailure( pD3D10Device->CreateRenderTargetView(pBB, &rtd, &pView),
    "Couldn't create view" );
pD3D10Device->OMSetRenderTargets(1, &pView, 0);
```

## Care and Feeding of the Swap Chain

A swap chain is a gentle beast. Once you've rendered your image, call `IDXGISwapChain::Present` and go render the next image. That is the extent of your responsibility.

If you've previously called `IDXGIFactory::MakeWindowAssociation`, the user can press the Alt-Enter key combination and DXGI will transition your application between windowed and fullscreen mode . Or you can disable Alt-Enter by not calling `IDXGIFactory::MakeWindowAssociation`.  
`IDXGIFactory::MakeWindowAssociation` is recommended, because a standard control mechanism for the user is strongly desired.

While you don't have to write any more code than has been described, a few simple steps can make your application more responsive. The most important consideration is the resizing of the swap chain's buffers in response to the resizing of the output window. Naturally, the application's best route is to respond to `WM_SIZE`, and call `IDXGISwapChain::ResizeBuffers`, passing the size contained in the message's parameters. This behavior obviously makes your application respond well to the user when he or she drags the window's borders, but it is also exactly what enables a smooth fullscreen transition. Your window will receive a `WM_SIZE` message whenever such a transition happens, and calling `IDXGISwapChain::ResizeBuffers` is the swap chain's chance to re-allocate the buffers' storage for optimal presentation. This is why the application is required to release any references it has on the existing buffers before it calls `IDXGISwapChain::ResizeBuffers`.

Failure to call `IDXGISwapChain::ResizeBuffers` in response to switching to fullscreen (most naturally, in response to `WM_SIZE`), can preclude the optimization of flipping, wherein DXGI can simply swap which buffer is being displayed, rather than copying a screen's full of data around.

`IDXGISwapChain::Present` will inform you if your output window is entirely occluded via `DXGI_STATUS_OCCLUDED`. When this occurs, it is recommended that your application go into standby mode (by calling `IDXGISwapChain::Present` with `DXGI_PRESENT_TEST`) since resources used to render the frame are wasted. Using `DXGI_PRESENT_TEST` will prevent any data from being presented while still

performing the occlusion check. Once IDXGISwapChain::Present returns S\_OK, you should exit standby mode; do not use the return code to switch to standby mode as doing so can leave the swapchain unable to relinquish fullscreen mode.

### Choosing the DXGI Output and Size

By default, DXGI chooses the output that contains most of the client area of the window. This is the only option available to DXGI when it goes fullscreen itself in response to alt-enter. If the application chooses to go fullscreen by itself, then it can call IDXGISwapChain::SetFullscreenState and pass an explicit IDXGIOOutput (or NULL, if the application is happy to let DXGI decide).

To resize the output while either fullscreen or windowed, then it's advisable to call IDXGISwapChain::ResizeTarget, since this method resizes the target window also. Since the target window is resized, WM\_SIZE is sent, and your code will naturally call IDXGISwapChain::ResizeBuffers in response. It's thus a waste of effort to resize your buffers, and then subsequently resize the target.

### Mode Switches

The DXGI swap chain might change the display mode of an output when making a full-screen transition. To enable the automatic display mode change, you must specify DXGI\_SWAP\_CHAIN\_FLAG\_ALLOW\_MODE\_SWITCH in the swap-chain description. If the display mode automatically changes, DXGI will choose the most modest mode (size and resolution will not change, but the color depth may). Resizing swap chain buffers will not cause a mode switch. The swap chain makes an implicit promise that if you choose a back buffer that exactly matches a display mode supported by the target output, then it will switch to that display mode when entering fullscreen mode on that output. Consequently, you choose a display mode by choosing your back buffer size and format.