

Contents

[Direct3D 11 Graphics](#)

[How to Use Direct3D 11](#)

[How To: Create a Reference Device](#)

[How To: Create a WARP Device](#)

[How To: Create a Swap Chain](#)

[How To: Enumerate Adapters](#)

[How To: Get Adapter Display Modes](#)

[How To: Create a Device and Immediate Context](#)

[How To: Get the Device Feature Level](#)

[How to: Create a Vertex Buffer](#)

[How to: Create an Index Buffer](#)

[How to: Create a Constant Buffer](#)

[How to: Create a Texture](#)

[How to: Initialize a Texture Programmatically](#)

[How to: Initialize a Texture From a File](#)

[How to: Use dynamic resources](#)

[How To: Create a Compute Shader](#)

[How To: Design a Hull Shader](#)

[How To: Create a Hull Shader](#)

[How To: Initialize the Tessellator Stage](#)

[How To: Design a Domain Shader](#)

[How To: Create a Domain Shader](#)

[How To: Compile a Shader](#)

[How to: Record a Command List](#)

[How to: Play Back a Command List](#)

[How To: Check for Driver Support](#)

[What's new in Direct3D 11](#)

[Direct3D 11 Features](#)

[Direct3D 11.1 Features](#)

[Direct3D 11.2 Features](#)

[Direct3D 11.3 Features](#)

[Conservative Rasterization](#)

[Default Texture Mapping](#)

[Rasterizer Order Views](#)

[Shader Specified Stencil Reference Value](#)

Typed Unordered Access View Loads
Unified Memory Architecture
Volume Tiled Resources
Direct3D 11.4 Features
Features Introduced In Previous Releases
What's New in the August 2009 Windows 7/Direct3D 11 SDK
What's new in the Nov 2008 Direct3D 11 Tech Preview

Programming Guide for Direct3D 11

Devices

Introduction to a Device in Direct3D 11
Software Layers
Using the debug layer to debug apps
Limitations Creating WARP and Reference Devices

Direct3D 11 on Downlevel Hardware

Direct3D feature levels
Exceptions
Compute Shaders on Downlevel Hardware
Preventing Unwanted NULL Pixel Shader SRVs
Using Direct3D 11 feature data to supplement Direct3D feature levels

Resources

Introduction to a Resource in Direct3D 11
Types of Resources
Resource Limits
Subresources
Buffers

Introduction to Buffers in Direct3D 11

Textures

Introduction To Textures in Direct3D 11
Texture Block Compression in Direct3D 11
BC6H Format
BC7 Format
BC7 Format Mode Reference

Floating-point rules

Tiled resources

Why are tiled resources needed?

Creating tiled resources

Mappings are into a tile pool
Tiled resource creation parameters
Address space available for tiled resources

- [Tile pool creation parameters](#)
- [Tiled resource cross process and device sharing](#)
- [Operations available on tiled resources](#)
- [Operations available on tile pools](#)
- [How a tiled resource's area is tiled](#)
- [Tiled Resource APIs](#)
- [Pipeline access to tiled resources](#)
 - [SRV behavior with non-mapped tiles](#)
 - [UAV behavior with non-mapped tiles](#)
 - [Rasterizer behavior with non-mapped tiles](#)
 - [Tile access limitations with duplicate mappings](#)
 - [Tiled resources texture sampling features](#)
 - [HLSL tiled resources exposure](#)
- [Tiled resources features tiers](#)
 - [Tier 1](#)
 - [Tier 2](#)
- [Graphics Pipeline](#)
 - [Input-Assembler Stage](#)
 - [Getting Started with the Input-Assembler Stage](#)
 - [Primitive Topologies](#)
 - [Using the Input-Assembler Stage without Buffers](#)
 - [Using System-Generated Values](#)
 - [Vertex Shader Stage](#)
 - [Tessellation Stages](#)
 - [Geometry Shader Stage](#)
 - [Stream-Output Stage](#)
 - [Getting Started with the Stream-Output Stage](#)
 - [Rasterizer Stage](#)
 - [Getting Started with the Rasterizer Stage](#)
 - [Rasterization Rules](#)
 - [Pixel Shader Stage](#)
 - [Output-Merger Stage](#)
 - [Configuring Depth-Stencil Functionality](#)
 - [Configuring Blending Functionality](#)
 - [Depth Bias](#)
- [Compute Shader Overview](#)
 - [New Resource Types](#)
 - [Accessing Resources](#)
 - [Atomic Functions](#)

Rendering

MultiThreading

[Introduction to Multithreading in Direct3D 11](#)

[Object Creation with Multithreading](#)

[Immediate and Deferred Rendering](#)

[Command List](#)

[Threading Differences between Direct3D Versions](#)

Multiple-Pass Rendering

Effects

[Organizing State in an Effect](#)

[Effect System Interfaces](#)

[Specializing Interfaces](#)

[Interfaces and Classes in Effects](#)

[Rendering an Effect](#)

[Compile an Effect](#)

[Create an Effect](#)

[Set Effect State](#)

[Apply a Technique](#)

[Cloning an Effect](#)

[Stream Out Syntax](#)

[Differences Between Effects 10 and Effects 11](#)

Migrating to Direct3D 11

Direct3D Video Interfaces

Direct3D 11, utilities, and effects reference

Direct3D 11 Reference

[Core reference](#)

[Core interfaces](#)

[ID3D11Asynchronous](#)

[ID3D11BlendState](#)

[ID3D11BlendState1](#)

[ID3D11CommandList](#)

[ID3D11Counter](#)

[ID3D11DepthStencilState](#)

[ID3D11Device](#)

[ID3D11Device1](#)

[ID3D11Device2](#)

[ID3D11Device3](#)

[ID3D11Device4](#)

[ID3D11Device5](#)

[ID3D11DeviceChild](#)
[ID3D11DeviceContext](#)
[ID3D11DeviceContext1](#)
[ID3D11DeviceContext2](#)
[ID3D11DeviceContext3](#)
[ID3D11DeviceContext4](#)
[ID3DDeviceContextState](#)
[ID3D11Fence](#)
[ID3D11InputLayout](#)
[ID3D11Multithread](#)
[ID3D11Predicate](#)
[ID3D11Query](#)
[ID3D11Query1](#)
[ID3D11RasterizerState](#)
[ID3D11RasterizerState1](#)
[ID3D11RasterizerState2](#)
[ID3D11SamplerState](#)

Core functions

[D3D11CreateDevice](#)
[D3D11CreateDeviceAndSwapChain](#)

Core structures

[D3D11_BLEND_DESC](#)
[D3D11_BLEND_DESC1](#)
[D3D11_BOX](#)
[D3D11_COUNTER_DESC](#)
[D3D11_COUNTER_INFO](#)
[D3D11_DEPTH_STENCIL_DESC](#)
[D3D11_DEPTH_STENCILOP_DESC](#)
[D3D11_DRAW_INDEXED_INSTANCED_INDIRECT_ARGS](#)
[D3D11_DRAW_INSTANCED_INDIRECT_ARGS](#)
[D3D11_FEATURE_DATA_ARCHITECTURE_INFO](#)
[D3D11_FEATURE_DATA_D3D9_OPTIONS](#)
[D3D11_FEATURE_DATA_D3D9_OPTIONS1](#)
[D3D11_FEATURE_DATA_D3D9_SHADOW_SUPPORT](#)
[D3D11_FEATURE_DATA_D3D9_SIMPLE_INSTANCING_SUPPORT](#)
[D3D11_FEATURE_DATA_D3D10_X_HARDWARE_OPTIONS](#)
[D3D11_FEATURE_DATA_D3D11_OPTIONS](#)
[D3D11_FEATURE_DATA_D3D11_OPTIONS1](#)
[D3D11_FEATURE_DATA_D3D11_OPTIONS2](#)

D3D11_FEATURE_DATA_D3D11_OPTIONS3
D3D11_FEATURE_DATA_D3D11_OPTIONS4
D3D11_FEATURE_DATA_D3D11_OPTIONS5
D3D11_FEATURE_DATA_DOUBLES
D3D11_FEATURE_DATA_FORMAT_SUPPORT
D3D11_FEATURE_DATA_FORMAT_SUPPORT2
D3D11_FEATURE_DATA_GPU_VIRTUAL_ADDRESS_SUPPORT
D3D11_FEATURE_DATA_MARKER_SUPPORT
D3D11_FEATURE_DATA_SHADER_CACHE
D3D11_FEATURE_DATA_SHADER_MIN_PRECISION_SUPPORT
D3D11_FEATURE_DATA_THREADING
D3D11_INPUT_ELEMENT_DESC
D3D11_QUERY_DATA_PIPELINE_STATISTICS
D3D11_QUERY_DATA_SO_STATISTICS
D3D11_QUERY_DATA_TIMESTAMP_DISJOINT
D3D11_QUERY_DESC
D3D11_QUERY_DESC1
D3D11_RASTERIZER_DESC
D3D11_RASTERIZER_DESC1
D3D11_RASTERIZER_DESC2
D3D11_RECT
D3D11_RENDER_TARGET_BLEND_DESC
D3D11_RENDER_TARGET_BLEND_DESC1
D3D11_SAMPLER_DESC
D3D11_SO_DECLARATION_ENTRY
D3D11_VIEWPORT

Core enumerations

D3D11_ASYNC_GETDATA_FLAG
D3D11_BLEND
D3D11_BLEND_OP
D3D11_CLEAR_FLAG
D3D11_COLOR_WRITE_ENABLE
D3D11_COMPARISON_FUNC
D3D11_CONSERVATIVE_RASTERIZATION_MODE
D3D11_CONSERVATIVE_RASTERIZATION_TIER
D3D11_CONTEXT_TYPE
D3D11_COPY_FLAGS
D3D11_COUNTER
D3D11_COUNTER_TYPE

D3D11_CREATE_DEVICE_FLAG
D3D11_1_CREATE_DEVICE_CONTEXT_STATE_FLAG
D3D11_CULL_MODE
D3D11_DEPTH_WRITE_MASK
D3D11_DEVICE_CONTEXT_TYPE
D3D11_FEATURE
D3D11_FENCE_FLAG
D3D11_FILL_MODE
D3D11_FILTER
D3D11_FILTER_TYPE
D3D11_FILTER_REDUCTION_TYPE
D3D11_FORMAT_SUPPORT
D3D11_FORMAT_SUPPORT2
D3D11_INPUT_CLASSIFICATION
D3D11_LOGIC_OP
D3D11_PRIMITIVE
D3D11_PRIMITIVE_TOPOLOGY
D3D11_QUERY
D3D11_QUERY_MISC_FLAG
D3D11_RAISE_FLAG
D3D11_SHADER_CACHE_SUPPORT_FLAGS
D3D11_SHADER_MIN_PRECISION_SUPPORT
D3D11_SHARED_RESOURCE_TIER
D3D11_STENCIL_OP
D3D11_TEXTURE_ADDRESS_MODE
D3D11_TEXTURECUBE_FACE
D3D11_TILED_RESOURCES_TIER

Layer Reference

Layer Interfaces

ID3D11Debug
ID3D11InfoQueue
ID3D11RefDefaultTrackingOptions
ID3D11RefTrackingOptions
ID3D11SwitchToRef
ID3D11TracingDevice

Layer Structures

D3D11_INFO_QUEUE_FILTER
D3D11_INFO_QUEUE_FILTER_DESC
D3D11_MESSAGE

Layer Enumerations

[D3D11_MESSAGE_CATEGORY](#)
[D3D11_MESSAGE_ID](#)
[D3D11_MESSAGE_SEVERITY](#)
[D3D11_RLDO_FLAGS](#)
[D3D11_SHADER_TRACKING_OPTIONS](#)
[D3D11_SHADER_TRACKING_RESOURCE_TYPE](#)

Resource Reference

Resource Interfaces

[ID3D11Buffer](#)
[ID3D11DepthStencilView](#)
[ID3D11RenderTargetView](#)
[ID3D11RenderTargetView1](#)
[ID3D11Resource](#)
[ID3D11ShaderResourceView](#)
[ID3D11ShaderResourceView1](#)
[ID3D11Texture1D](#)
[ID3D11Texture2D](#)
[ID3D11Texture2D1](#)
[ID3D11Texture3D](#)
[ID3D11Texture3D1](#)
[ID3D11UnorderedAccessView](#)
[ID3D11UnorderedAccessView1](#)
[ID3D11View](#)

Resource Functions

[D3D11CalcSubresource](#)

Resource Structures

[D3D11_BUFFER_DESC](#)
[D3D11_BUFFER_RTV](#)
[D3D11_BUFFER_SRV](#)
[D3D11_BUFFER_UAV](#)
[D3D11_BUFFEREX_SRV](#)
[D3D11_DEPTH_STENCIL_VIEW_DESC](#)
[D3D11_MAPPED_SUBRESOURCE](#)
[D3D11_PACKED_MIP_DESC](#)
[D3D11_RENDER_TARGET_VIEW_DESC](#)
[D3D11_RENDER_TARGET_VIEW_DESC1](#)
[D3D11_SHADER_RESOURCE_VIEW_DESC](#)
[D3D11_SHADER_RESOURCE_VIEW_DESC1](#)

D3D11_SUBRESOURCE_DATA
D3D11_SUBRESOURCE_TILING
D3D11_TEX1D_ARRAY_DSV
D3D11_TEX1D_ARRAY_RTV
D3D11_TEX1D_ARRAY_SRV
D3D11_TEX1D_ARRAY_UAV
D3D11_TEX1D_DSV
D3D11_TEX1D_RTV
D3D11_TEX1D_SRV
D3D11_TEX1D_UAV
D3D11_TEX2D_ARRAY_DSV
D3D11_TEX2D_ARRAY_RTV
D3D11_TEX2D_ARRAY_RTV1
D3D11_TEX2D_ARRAY_SRV
D3D11_TEX2D_ARRAY_SRV1
D3D11_TEX2D_ARRAY_UAV
D3D11_TEX2D_ARRAY_UAV1
D3D11_TEX2D_DSV
D3D11_TEX2D_RTV
D3D11_TEX2D_RTV1
D3D11_TEX2D_SRV
D3D11_TEX2D_SRV1
D3D11_TEX2D_UAV
D3D11_TEX2D_UAV1
D3D11_TEX2DMS_ARRAY_DSV
D3D11_TEX2DMS_ARRAY_RTV
D3D11_TEX2DMS_ARRAY_SRV
D3D11_TEX2DMS_DSV
D3D11_TEX2DMS_RTV
D3D11_TEX2DMS_SRV
D3D11_TEX3D_RTV
D3D11_TEX3D_SRV
D3D11_TEX3D_UAV
D3D11_TEXCUBE_ARRAY_SRV
D3D11_TEXCUBE_SRV
D3D11_TEXTURE1D_DESC
D3D11_TEXTURE2D_DESC
D3D11_TEXTURE2D_DESC1
D3D11_TEXTURE3D_DESC

[D3D11_TEXTURE3D_DESC1](#)
[D3D11_TILE_REGION_SIZE](#)
[D3D11_TILED_RESOURCE_COORDINATE](#)
[D3D11_TILE_SHAPE](#)
[D3D11_UNORDERED_ACCESS_VIEW_DESC](#)
[D3D11_UNORDERED_ACCESS_VIEW_DESC1](#)

Resource Enumerations

[D3D11_BIND_FLAG](#)
[D3D11_BUFFEREX_SRV_FLAG](#)
[D3D11_BUFFER_UAV_FLAG](#)
[D3D11_CHECK_MULTISAMPLE_QUALITY_LEVELS_FLAG](#)
[D3D11_CPU_ACCESS_FLAG](#)
[D3D11_DSV_DIMENSION](#)
[D3D11_DSV_FLAG](#)
[D3D11_MAP](#)
[D3D11_MAP_FLAG](#)
[D3D11_RESOURCE_DIMENSION](#)
[D3D11_RESOURCE_MISC_FLAG](#)
[D3D11_RTV_DIMENSION](#)
[D3D11_SRV_DIMENSION](#)
[D3D11_STANDARD_MULTISAMPLE_QUALITY_LEVELS](#)
[D3D11_TEXTURE_LAYOUT](#)
[D3D11_TILE_COPY_FLAG](#)
[D3D11_TILE_MAPPING_FLAG](#)
[D3D11_TILE_RANGE_FLAG](#)
[D3D11_UAV_DIMENSION](#)
[D3D11_USAGE](#)

Shader Reference

Shader Interfaces

[ID3D11ClassInstance](#)
[ID3D11ClassLinkage](#)
[ID3D11ComputeShader](#)
[ID3D11DomainShader](#)
[ID3D11FunctionLinkingGraph](#)
[ID3D11FunctionReflection](#)
[ID3D11FunctionParameterReflection](#)
[ID3D11GeometryShader](#)
[ID3D11HullShader](#)
[ID3D11LibraryReflection](#)

[ID3D11Linker](#)

[ID3D11LinkingNode](#)

[ID3D11Module](#)

[ID3D11ModuleInstance](#)

[ID3D11PixelShader](#)

[ID3D11ShaderReflection](#)

[ID3D11ShaderReflectionConstantBuffer](#)

[ID3D11ShaderReflectionType](#)

[ID3D11ShaderReflectionVariable](#)

[ID3D11ShaderTrace](#)

[ID3D11ShaderTraceFactory](#)

[ID3D11VertexShader](#)

Shader Functions

[D3DDisassemble11Trace](#)

Shader Structures

[D3D11_CLASS_INSTANCE_DESC](#)

[D3D11_COMPUTE_SHADER_TRACE_DESC](#)

[D3D11_DOMAIN_SHADER_TRACE_DESC](#)

[D3D11_FUNCTION_DESC](#)

[D3D11_GEOMETRY_SHADER_TRACE_DESC](#)

[D3D11_HULL_SHADER_TRACE_DESC](#)

[D3D11_LIBRARY_DESC](#)

[D3D11_PARAMETER_DESC](#)

[D3D11_PIXEL_SHADER_TRACE_DESC](#)

[D3D11_SHADER_BUFFER_DESC](#)

[D3D11_SHADER_DESC](#)

[D3D11_SHADER_INPUT_BIND_DESC](#)

[D3D11_SHADER_TRACE_DESC](#)

[D3D11_SHADER_TYPE_DESC](#)

[D3D11_SHADER_VARIABLE_DESC](#)

[D3D11_SIGNATURE_PARAMETER_DESC](#)

[D3D11_TRACE_REGISTER](#)

[D3D11_TRACE_STATS](#)

[D3D11_TRACE_STEP](#)

[D3D11_TRACE_VALUE](#)

[D3D11_VERTEX_SHADER_TRACE_DESC](#)

Shader Enumerations

[D3D11_CBUFFER_TYPE](#)

[D3D_PARAMETER_FLAGS](#)

[D3D11_RESOURCE_RETURN_TYPE](#)
[D3D11_SHADER_TYPE](#)
[D3D11_SHADER_VERSION_TYPE](#)
[D3D11_TESSELLATOR_DOMAIN](#)
[D3D11_TESSELLATOR_PARTITIONING](#)
[D3D11_TESSELLATOR_OUTPUT_PRIMITIVE](#)
[D3D11_TRACE_GS_INPUT_PRIMITIVE](#)
[D3D11_TRACE_REGISTER_TYPE](#)

[10Level9 Reference](#)

[10Level9 ID3D11Device Methods](#)
[10Level9 ID3D11DeviceContext Methods](#)

[Direct3D 11 Return Codes](#)

[Common Version Reference](#)

[Common Version Interfaces](#)

[ID3D10Blob](#)
[ID3DBlob](#)
[ID3DDestructionNotifier](#)
[ID3DInclude](#)
[ID3DUserDefinedAnnotation](#)

[Common Version Structures](#)

[D3D_SHADER_MACRO](#)

[Common Version Enumerations](#)

[D3D_CBUFFER_TYPE](#)
[D3D_DRIVER_TYPE](#)
[D3D_FEATURE_LEVEL](#)
[D3D_INCLUDE_TYPE](#)
[D3D_INTERPOLATION_MODE](#)
[D3D_MIN_PRECISION](#)
[D3D_NAME](#)
[D3D_PRIMITIVE](#)

[D3D_PRIMITIVE_TOPOLOGY](#)
[D3D_REGISTER_COMPONENT_TYPE](#)
[D3D_RESOURCE_RETURN_TYPE](#)
[D3D_SHADER_CBUFFER_FLAGS](#)
[D3D_SHADER_INPUT_FLAGS](#)
[D3D_SHADER_INPUT_TYPE](#)
[D3D_SHADER_VARIABLE_CLASS](#)
[D3D_SHADER_VARIABLE_FLAGS](#)
[D3D_SHADER_VARIABLE_TYPE](#)

[D3D_SRV_DIMENSION](#)

[D3D_TESSELLATOR_DOMAIN](#)

[D3D_TESSELLATOR_OUTPUT_PRIMITIVE](#)

[D3D_TESSELLATOR_PARTITIONING](#)

CD3D11 Helper Structures

[CD3D11_RECT](#)

[CD3D11_RECT\(\) constructor](#)

[CD3D11_RECT\(constD3D11_RECT\) constructor](#)

[CD3D11_RECT\(LONG, LONG, LONG, LONG\) constructor](#)

[~CD3D11_RECT\(\) destructor](#)

[operator const D3D11_RECT&\(\) method](#)

[CD3D11_BOX](#)

[CD3D11_BOX\(\) constructor](#)

[CD3D11_BOX\(constD3D11_BOX\) constructor](#)

[CD3D11_BOX\(LONG, LONG, LONG, LONG, LONG, LONG\) constructor](#)

[~CD3D11_BOX\(\) destructor](#)

[operator const D3D11_BOX&\(\) method](#)

[CD3D11_DEPTH_STENCIL_DESC](#)

[CD3D11_DEPTH_STENCIL_DESC\(\) constructor](#)

[CD3D11_DEPTH_STENCIL_DESC\(constD3D11_DEPTH_STENCIL_DESC\) constructor](#)

[CD3D11_DEPTH_STENCIL_DESC\(CD3D11_DEFAULT\) constructor](#)

[CD3D11_DEPTH_STENCIL_DESC\(BOOL, D3D11_DEPTH_WRITE_MASK,
D3D11_COMPARISON_FUNC, BOOL, UINT8, UINT8, D3D11_STENCIL_OP,
D3D11_STENCIL_OP, D3D11_STENCIL_OP, D3D11_COMPARISON_FUNC, D3D11_STENCIL_OP,
D3D11_STENCIL_OP, D3D11_STENCIL_OP, D3D11_COMPARISON_FUNC\) constructor](#)

[~CD3D11_DEPTH_STENCIL_DESC\(\) destructor](#)

[operator const D3D11_DEPTH_STENCIL_DESC&\(\) method](#)

[CD3D11_BLEND_DESC](#)

[CD3D11_BLEND_DESC\(\) constructor](#)

[CD3D11_BLEND_DESC\(constD3D11_BLEND_DESC\) constructor](#)

[CD3D11_BLEND_DESC\(CD3D11_DEFAULT\) constructor](#)

[~CD3D11_BLEND_DESC\(\) destructor](#)

[operator const D3D11_BLEND_DESC&\(\) method](#)

[CD3D11_RASTERIZER_DESC](#)

[CD3D11_RASTERIZER_DESC\(\) constructor](#)

[CD3D11_RASTERIZER_DESC\(constD3D11_RASTERIZER_DESC\) constructor](#)

[CD3D11_RASTERIZER_DESC\(CD3D11_DEFAULT\) constructor](#)

[CD3D11_RASTERIZER_DESC\(D3D11_FILL_MODE, D3D11_CULL_MODE, BOOL, INT, FLOAT,
FLOAT, BOOL, BOOL, BOOL, BOOL\) constructor](#)

[~CD3D11_RASTERIZER_DESC\(\) destructor](#)

operator const D3D11_RASTERIZER_DESC&() method

CD3D11_BUFFER_DESC

- CD3D11_BUFFER_DESC() constructor
- CD3D11_BUFFER_DESC(const D3D11_BUFFER_DESC) constructor
- CD3D11_BUFFER_DESC(UINT, D3D11_USAGE, UINT, UINT, UINT, D3D11_USAGE, UINT) constructor
- ~CD3D11_BUFFER_DESC() destructor

operator const D3D11_BUFFER_DESC&() method

CD3D11_TEXTURE1D_DESC

- CD3D11_TEXTURE1D_DESC() constructor
- CD3D11_TEXTURE1D_DESC(const D3D11_TEXTURE1D_DESC&) constructor
- CD3D11_TEXTURE1D_DESC(DXGI_FORMAT, UINT, UINT, UINT, D3D11_USAGE, UINT, UINT) constructor
- ~CD3D11_TEXTURE1D_DESC() destructor

operator const D3D11_TEXTURE1D_DESC&() method

CD3D11_TEXTURE2D_DESC

- CD3D11_TEXTURE2D_DESC() constructor
- CD3D11_TEXTURE2D_DESC(const D3D11_TEXTURE2D_DESC&) constructor
- CD3D11_TEXTURE2D_DESC(DXGI_FORMAT, UINT, UINT, UINT, D3D11_USAGE, UINT, UINT, UINT, UINT) constructor
- ~CD3D11_TEXTURE2D_DESC() destructor

operator const D3D11_TEXTURE2D_DESC&() method

CD3D11_TEXTURE3D_DESC

- CD3D11_TEXTURE3D_DESC() constructor
- CD3D11_TEXTURE3D_DESC(const D3D11_TEXTURE3D_DESC&) constructor
- CD3D11_TEXTURE3D_DESC(DXGI_FORMAT, UINT, UINT, D3D11_USAGE, UINT, D3D11_USAGE, UINT, UINT) constructor
- ~CD3D11_TEXTURE3D_DESC() destructor

operator const D3D11_TEXTURE3D_DESC&() method

CD3D11_SHADER_RESOURCE_VIEW_DESC

- CD3D11_SHADER_RESOURCE_VIEW_DESC() constructor
- CD3D11_SHADER_RESOURCE_VIEW_DESC(const D3D11_SHADER_RESOURCE_VIEW_DESC&) constructor
- CD3D11_SHADER_RESOURCE_VIEW_DESC(D3D11_SRV_DIMENSION, DXGI_FORMAT, D3D11_SRV_DIMENSION, D3D11_SRV_DIMENSION, DXGI_FORMAT, D3D11_SRV_DIMENSION, D3D11_SRV_DIMENSION) constructor
- CD3D11_SHADER_RESOURCE_VIEW_DESC(ID3D11Buffer*, DXGI_FORMAT, D3D11_SRV_DIMENSION, D3D11_SRV_DIMENSION, DXGI_FORMAT, D3D11_SRV_DIMENSION, D3D11_SRV_DIMENSION) constructor
- CD3D11_SHADER_RESOURCE_VIEW_DESC(ID3D11Texture1D*, D3D11_SRV_DIMENSION, DXGI_FORMAT, D3D11_SRV_DIMENSION, D3D11_SRV_DIMENSION, DXGI_FORMAT, D3D11_SRV_DIMENSION, D3D11_SRV_DIMENSION) constructor
- CD3D11_SHADER_RESOURCE_VIEW_DESC(ID3D11Texture2D*, D3D11_SRV_DIMENSION, DXGI_FORMAT, D3D11_SRV_DIMENSION, D3D11_SRV_DIMENSION, DXGI_FORMAT, D3D11_SRV_DIMENSION, D3D11_SRV_DIMENSION) constructor
- CD3D11_SHADER_RESOURCE_VIEW_DESC(ID3D11Texture3D*, DXGI_FORMAT, D3D11_SRV_DIMENSION, D3D11_SRV_DIMENSION, DXGI_FORMAT, D3D11_SRV_DIMENSION, D3D11_SRV_DIMENSION, DXGI_FORMAT) constructor

constructor

`~CD3D11_SHADER_RESOURCE_VIEW_DESC()` destructor

`operator const D3D11_SHADER_RESOURCE_VIEW_DESC&()` method

`CD3D11_RENDER_TARGET_VIEW_DESC`

`CD3D11_RENDER_TARGET_VIEW_DESC()` constructor

`CD3D11_RENDER_TARGET_VIEW_DESC(const D3D11_RENDER_TARGET_VIEW_DESC&)` constructor

`CD3D11_RENDER_TARGET_VIEW_DESC(D3D11_RTV_DIMENSION, DXGI_FORMAT, UINT, UINT, UINT)` constructor

`CD3D11_RENDER_TARGET_VIEW_DESC(ID3D11Buffer*, DXGI_FORMAT, UINT, UINT)` constructor

`CD3D11_RENDER_TARGET_VIEW_DESC(ID3D11Texture1D*, D3D11_RTV_DIMENSION, DXGI_FORMAT, UINT, UINT, UINT)` constructor

`CD3D11_RENDER_TARGET_VIEW_DESC(ID3D11Texture2D*, D3D11_RTV_DIMENSION, DXGI_FORMAT, UINT, UINT, UINT)` constructor

`CD3D11_RENDER_TARGET_VIEW_DESC(ID3D11Texture3D*, DXGI_FORMAT, UINT, UINT, UINT)` constructor

`~CD3D11_RENDER_TARGET_VIEW_DESC()` destructor

`operator const D3D11_RENDER_TARGET_VIEW_DESC&()` method

`CD3D11_VIEWPORT`

`CD3D11_VIEWPORT()` constructor

`CD3D11_VIEWPORT(const D3D11_VIEWPORT&)` constructor

`CD3D11_VIEWPORT(FLOAT, FLOAT, FLOAT, FLOAT, FLOAT, FLOAT)` constructor

`CD3D11_VIEWPORT(ID3D11Buffer*, ID3D11RenderTargetView*, FLOAT, FLOAT, FLOAT)` constructor

`CD3D11_VIEWPORT(ID3D11Texture1D*, ID3D11RenderTargetView*, FLOAT, FLOAT, FLOAT)` constructor

`CD3D11_VIEWPORT(ID3D11Texture2D*, ID3D11RenderTargetView*, FLOAT, FLOAT, FLOAT)` constructor

`CD3D11_VIEWPORT(ID3D11Texture3D*, ID3D11RenderTargetView*, FLOAT, FLOAT, FLOAT)` constructor

`~CD3D11_VIEWPORT()` destructor

`operator const D3D11_VIEWPORT&()` method

`CD3D11_DEPTH_STENCIL_VIEW_DESC`

`CD3D11_DEPTH_STENCIL_VIEW_DESC()` constructor

`CD3D11_DEPTH_STENCIL_VIEW_DESC(const D3D11_DEPTH_STENCIL_VIEW_DESC&)` constructor

`CD3D11_DEPTH_STENCIL_VIEW_DESC(D3D11_DSV_DIMENSION, DXGI_FORMAT, UINT, UINT, UINT)` constructor

`CD3D11_DEPTH_STENCIL_VIEW_DESC(ID3D11Texture1D*, D3D11_DSV_DIMENSION, DXGI_FORMAT, UINT, UINT, UINT, UINT)` constructor

`CD3D11_DEPTH_STENCIL_VIEW_DESC(ID3D11Texture2D*, D3D11_DSV_DIMENSION,`

`DXGI_FORMAT, UINT, UINT, UINT, UINT)` constructor
~`CD3D11_DEPTH_STENCIL_VIEW_DESC()` destructor
`operator const D3D11_DEPTH_STENCIL_VIEW_DESC&()` method

`CD3D11_UNORDERED_ACCESS_VIEW_DESC`

`CD3D11_UNORDERED_ACCESS_VIEW_DESC()` constructor

`CD3D11_UNORDERED_ACCESS_VIEW_DESC(const D3D11_UNORDERED_ACCESS_VIEW_DESC&)` constructor
`CD3D11_UNORDERED_ACCESS_VIEW_DESC(D3D11_UAV_DIMENSION, DXGI_FORMAT, UINT, UINT, UINT, UINT)` constructor
`CD3D11_UNORDERED_ACCESS_VIEW_DESC(ID3D11Buffer*, DXGI_FORMAT, UINT, UINT, UINT)` constructor
`CD3D11_UNORDERED_ACCESS_VIEW_DESC(ID3D11Texture1D*, D3D11_UAV_DIMENSION, DXGI_FORMAT, UINT, UINT, UINT)` constructor
`CD3D11_UNORDERED_ACCESS_VIEW_DESC(ID3D11Texture2D*, D3D11_UAV_DIMENSION, DXGI_FORMAT, UINT, UINT, UINT)` constructor
`CD3D11_UNORDERED_ACCESS_VIEW_DESC(ID3D11Texture3D*, DXGI_FORMAT, UINT, UINT, UINT)` constructor
~`CD3D11_UNORDERED_ACCESS_VIEW_DESC()` destructor
`operator const D3D11_UNORDERED_ACCESS_VIEW_DESC&()` method

`CD3D11_SAMPLER_DESC`

`CD3D11_SAMPLER_DESC()` constructor
`CD3D11_SAMPLER_DESC(const D3D11_SAMPLER_DESC&)` constructor
`CD3D11_SAMPLER_DESC(CD3D11_DEFAULT)` constructor
~`CD3D11_SAMPLER_DESC()` destructor
`operator const D3D11_SAMPLER_DESC&()` method

`CD3D11_QUERY_DESC`

`CD3D11_QUERY_DESC()` constructor
`CD3D11_QUERY_DESC(const D3D11_QUERY_DESC&)` constructor
`CD3D11_QUERY_DESC(D3D11_QUERY, UINT)` constructor
~`CD3D11_QUERY_DESC()` destructor
`operator const D3D11_QUERY_DESC&()` method

`CD3D11_COUNTER_DESC`

`CD3D11_COUNTER_DESC()` constructor
`CD3D11_COUNTER_DESC(const D3D11_COUNTER_DESC&)` constructor
`CD3D11_COUNTER_DESC(D3D11_COUNTER, UINT)` constructor
~`CD3D11_COUNTER_DESC()` method
`operator const D3D11_COUNTER_DESC&()` method

[ID3DX11FFT](#)
[AttachBuffersAndPrecompute](#)
[ForwardTransform](#)
[GetForwardScale](#)
[GetInverseScale](#)
[InverseTransform](#)
[SetForwardScale](#)
[SetInverseScale](#)

[ID3DX11Scan](#)
[Multiscan](#)
[Scan](#)
[SetScanDirection](#)

[ID3DX11SegmentedScan](#)
[SegScan](#)
[SetScanDirection](#)

[D3DCSX 11 Functions](#)
[D3DX11CreateScan](#)
[D3DX11CreateSegmentedScan](#)
[D3DX11CreateFFT](#)
[D3DX11CreateFFT1DComplex](#)
[D3DX11CreateFFT1DReal](#)
[D3DX11CreateFFT2DComplex](#)
[D3DX11CreateFFT2DReal](#)
[D3DX11CreateFFT3DComplex](#)
[D3DX11CreateFFT3DReal](#)

[D3DCSX 11 Structures](#)
[D3DX11_FFT_BUFFER_INFO](#)
[D3DX11_FFT_DESC](#)

[D3DCSX 11 Enumerations](#)
[D3DX11_FFT_CREATE_FLAG](#)
[D3DX11_FFT_DATA_TYPE](#)
[D3DX11_FFT_DIM_MASK](#)
[D3DX11_SCAN_DATA_TYPE](#)
[D3DX11_SCAN_DIRECTION](#)
[D3DX11_SCAN_OPCODE](#)

[D3DX 11 Reference](#)
[D3DX Interfaces](#)
[ID3DX11DataLoader](#)
[Decompress](#)

Destroy
Load

ID3DX11DataProcessor

CreateDeviceObject
Destroy
Process

ID3DX11ThreadPump

AddWorkItem
GetQueueStatus
GetWorkItemCount
ProcessDeviceWorkItems
PurgeAllItems
WaitForAllItems

D3DX Functions

D3DX11CompileFromFile
D3DX11CompileFromMemory
D3DX11CompileFromResource
D3DX11ComputeNormalMap
D3DX11CreateAsyncCompilerProcessor
D3DX11CreateAsyncFileLoader
D3DX11CreateAsyncMemoryLoader
D3DX11CreateAsyncResourceLoader
D3DX11CreateAsyncShaderPreprocessProcessor
D3DX11CreateAsyncTextureInfoProcessor
D3DX11CreateAsyncTextureProcessor
D3DX11CreateAsyncShaderResourceViewProcessor
D3DX11CreateShaderResourceViewFromFile
D3DX11CreateShaderResourceViewFromMemory
D3DX11CreateShaderResourceViewFromResource
D3DX11CreateTextureFromFile
D3DX11CreateTextureFromMemory
D3DX11CreateTextureFromResource
D3DX11CreateThreadPump
D3DX11FilterTexture
D3DX11GetImageInfoFromFile
D3DX11GetImageInfoFromMemory
D3DX11GetImageInfoFromResource
D3DX11LoadTextureFromTexture
D3DX11PreprocessShaderFromFile

[D3DX11PreprocessShaderFromMemory](#)
[D3DX11PreprocessShaderFromResource](#)
[D3DX11SaveTextureToFile](#)
[D3DX11SaveTextureToMemory](#)
[D3DX11SHProjectCubeMap](#)
[D3DX11UnsetAllDeviceObjects](#)

D3DX Structures

[D3DX11_IMAGE_INFO](#)
[D3DX11_IMAGE_LOAD_INFO](#)
[D3DX11_TEXTURE_LOAD_INFO](#)

D3DX Enumerations

[D3DX11_CHANNEL_FLAG](#)
[D3DX11_ERR](#)
[D3DX11_FILTER_FLAG](#)
[D3DX11_IMAGE_FILE_FORMAT](#)
[D3DX11_NORMALMAP_FLAG](#)
[D3DX11_SAVE_TEXTURE_FLAG](#)

Effects 11 Reference

Effects 11 Interfaces

[ID3DX11Effect](#)
[CloneEffect](#)
[GetClassLinkage](#)
[GetConstantBufferByIndex](#)
[GetConstantBufferByName](#)
[GetDesc](#)
[GetDevice](#)
[GetGroupByIndex](#)
[GetGroupByName](#)
[GetTechniqueByIndex](#)
[GetTechniqueByName](#)
[GetVariableByIndex](#)
[GetVariableByName](#)
[GetVariableBySemantic](#)
[IsOptimized](#)
[IsValid](#)
[Optimize](#)

[ID3DX11EffectBlendVariable](#)
[GetBackingStore](#)
[GetBlendState](#)

[SetBlendState](#)
[UndoSetBlendState](#)
[ID3DX11EffectClassInstanceVariable](#)
 [GetClassInstance](#)
[ID3DX11EffectConstantBuffer](#)
 [GetConstantBuffer](#)
 [GetTextureBuffer](#)
 [SetConstantBuffer](#)
 [SetTextureBuffer](#)
 [UndoSetConstantBuffer](#)
 [UndoSetTextureBuffer](#)
[ID3DX11EffectDepthStencilVariable](#)
 [GetBackingStore](#)
 [GetDepthStencilState](#)
 [SetDepthStencilState](#)
 [UndoSetDepthStencilState](#)
[ID3DX11EffectDepthStencilViewVariable](#)
 [GetDepthStencil](#)
 [GetDepthStencilArray](#)
 [SetDepthStencil](#)
 [SetDepthStencilArray](#)
[ID3DX11EffectGroup](#)
 [GetAnnotationByIndex](#)
 [GetAnnotationByName](#)
 [GetDesc](#)
 [GetTechniqueByIndex](#)
 [GetTechniqueByName](#)
 [IsValid](#)
[ID3DX11EffectInterfaceVariable](#)
 [GetClassInstance](#)
 [SetClassInstance](#)
[ID3DX11EffectMatrixVariable](#)
 [GetMatrix](#)
 [GetMatrixArray](#)
 [GetMatrixTranspose](#)
 [GetMatrixTransposeArray](#)
 [SetMatrix](#)
 [SetMatrixArray](#)
 [SetMatrixTranspose](#)

[SetMatrixTransposeArray](#)
[ID3DX11EffectPass](#)
 [Apply](#)
 [ComputeStateBlockMask](#)
 [GetAnnotationByIndex](#)
 [GetAnnotationByName](#)
 [GetComputeShaderDesc](#)
 [GetDesc](#)
 [GetDomainShaderDesc](#)
 [GetGeometryShaderDesc](#)
 [GetHullShaderDesc](#)
 [GetPixelShaderDesc](#)
 [GetVertexShaderDesc](#)
 [IsValid](#)
[ID3DX11EffectRasterizerVariable](#)
 [GetBackingStore](#)
 [GetRasterizerState](#)
 [SetRasterizerState](#)
 [UndoSetRasterizerState](#)
[ID3DX11EffectRenderTargetViewVariable](#)
 [GetRenderTarget](#)
 [GetRenderTargetArray](#)
 [SetRenderTarget](#)
 [SetRenderTargetArray](#)
[ID3DX11EffectSamplerVariable](#)
 [GetBackingStore](#)
 [GetSampler](#)
 [SetSampler](#)
 [UndoSetSampler](#)
[ID3DX11EffectScalarVariable](#)
 [GetBool](#)
 [GetBoolArray](#)
 [GetFloat](#)
 [GetFloatArray](#)
 [GetInt](#)
 [GetIntArray](#)
 [SetBool](#)
 [SetBoolArray](#)
 [SetFloat](#)

[SetFloatArray](#)
[SetInt](#)
[SetIntArray](#)
[ID3DX11EffectShaderResourceVariable](#)
 [GetResource](#)
 [GetResourceArray](#)
 [SetResource](#)
 [SetResourceArray](#)
[ID3DX11EffectShaderVariable](#)
 [GetComputeShader](#)
 [GetDomainShader](#)
 [GetGeometryShader](#)
 [GetHullShader](#)
 [GetInputSignatureElementDesc](#)
 [GetOutputSignatureElementDesc](#)
 [GetPatchConstantSignatureElementDesc](#)
 [GetPixelShader](#)
 [GetShaderDesc](#)
 [GetVertexShader](#)
[ID3DX11EffectStringVariable](#)
 [GetString](#)
 [GetStringArray](#)
[ID3DX11EffectTechnique](#)
 [ComputeStateBlockMask](#)
 [GetAnnotationByIndex](#)
 [GetAnnotationByName](#)
 [GetDesc](#)
 [GetPassByIndex](#)
 [GetPassByName](#)
 [IsValid](#)
[ID3DX11EffectType](#)
 [GetDesc](#)
 [GetMemberName](#)
 [GetMemberSemantic](#)
 [GetMemberTypeByIndex](#)
 [GetMemberTypeByName](#)
 [GetMemberTypeBySemantic](#)
 [IsValid](#)
[ID3DX11EffectUnorderedAccessViewVariable](#)

[GetUnorderedAccessView](#)
[GetUnorderedAccessViewArray](#)
[SetUnorderedAccessView](#)
[SetUnorderedAccessViewArray](#)

ID3DX11EffectVariable

[AsBlend](#)
[AsClassInstance](#)
[AsConstantBuffer](#)
[AsDepthStencil](#)
[AsDepthStencilView](#)
[AsInterface](#)
[AsMatrix](#)
[AsRasterizer](#)
[AsRenderTargetView](#)
[AsSampler](#)
[AsScalar](#)
[AsShader](#)
[AsShaderResource](#)
[AsString](#)
[AsUnorderedAccessView](#)
[AsVector](#)
[GetAnnotationByIndex](#)
[GetAnnotationByName](#)
[GetDesc](#)
[GetElement](#)
[GetMemberByIndex](#)
[GetMemberByName](#)
[GetMemberBySemantic](#)
[GetParentConstantBuffer](#)
[GetRawValue](#)
[GetType](#)
[IsValid](#)
[SetRawValue](#)

ID3DX11EffectVectorVariable

[GetBoolVector](#)
[GetBoolVectorArray](#)
[GetFloatVector](#)
[GetFloatVectorArray](#)
[GetIntVector](#)

[GetIntVectorArray](#)

[SetBoolVector](#)

[SetBoolVectorArray](#)

[SetFloatVector](#)

[SetFloatVectorArray](#)

[SetIntVector](#)

[SetIntVectorArray](#)

Effects 11 Functions

[D3DX11CreateEffectFromMemory](#)

Effects 11 Structures

[D3DX11_EFFECT_DESC](#)

[D3DX11_EFFECT_SHADER_DESC](#)

[D3DX11_EFFECT_TYPE_DESC](#)

[D3DX11_EFFECT_VARIABLE_DESC](#)

[D3DX11_GROUP_DESC](#)

[D3DX11_PASS_DESC](#)

[D3DX11_PASS_SHADER_DESC](#)

[D3DX11_STATE_BLOCK_MASK](#)

[D3DX11_TECHNIQUE_DESC](#)

Effect Format

[Effect Variable Syntax](#)

[Annotation Syntax](#)

[Effect Function Syntax](#)

[Effect Technique Syntax](#)

[Effect State Groups](#)

[Effect Group Syntax](#)

Direct3D 11 Graphics

11/2/2020 • 2 minutes to read • [Edit Online](#)

You can use Microsoft Direct3D 11 graphics to create 3-D graphics for games and scientific and desktop applications.

This section contains information about programming with Direct3D 11 graphics.

For more information, see [Direct3D 11 Features](#).

Supported runtime environments	Windows/C++
Recommended programming languages	C/C++
Minimum supported Client	Windows 7
Minimum supported Server	Windows Server 2008 R2
TOPIC	DESCRIPTION
How to Use Direct3D 11	This section demonstrates how to use the Direct3D 11 API to accomplish several common tasks.
What's new in Direct3D 11	This section describes features added in Direct3D 11, Direct3D 11.1, and Direct3D 11.2.
Programming Guide for Direct3D 11	The programming guide contains information about how to use the Direct3D 11 programmable pipeline to create realtime 3D graphics for games as well as scientific and desktop applications.
Direct3D 11 Reference	This section contains the reference pages for Direct3D 11-based graphics programming.

In addition to Direct3D 11 being supported by Windows 7 and later and Windows Server 2008 R2 and later, Direct3D 11 is available down-level for Windows Vista with Service Pack 2 (SP2) and Windows Server 2008 by downloading [KB 971644](#) and [KB 971512](#).

For info about new Direct3D 11.1 features that are included with Windows 8, Windows Server 2012, and are partially available on Windows 7 and Windows Server 2008 R2 via the [Platform Update for Windows 7](#), see [Direct3D 11.1 Features](#) and the [Platform Update for Windows 7](#).

How to Use Direct3D 11

11/2/2020 • 2 minutes to read • [Edit Online](#)

This section demonstrates how to use the Microsoft Direct3D 11 API to accomplish several common tasks.

TOPIC	DESCRIPTION
How To: Create a Reference Device	This topic shows how to create a reference device that implements a highly accurate, software implementation of the runtime.
How To: Create a WARP Device	This topic shows how to create a WARP device that implements a high speed software rasterizer.
How To: Create a Swap Chain	This topic show how to create a swap chain that encapsulates two or more buffers that are used for rendering and display.
How To: Enumerate Adapters	This topic shows how to use Microsoft DirectX Graphics Infrastructure (DXGI) to enumerate the available graphics adapters on a computer.
How To: Get Adapter Display Modes	This topic shows how to use DXGI to get the valid display modes associated with an adapter.
How To: Create a Device and Immediate Context	This topics shows how to initialize a device .
How To: Get the Device Feature Level	This topics shows how to get the highest feature level supported by a device .
How to: Create a Vertex Buffer	This topic shows how to initialize a static vertex buffer , that is, a vertex buffer that does not change.
How to: Create an Index Buffer	This topic shows how to initialize an index buffer in preparation for rendering.
How to: Create a Constant Buffer	This topic shows how to initialize a constant buffer in preparation for rendering.
How to: Create a Texture	This topic shows how to create a texture.
How to: Initialize a Texture Programmatically	This topic has several examples showing how to initialize textures that are created with different types of usages.
How to: Initialize a Texture From a File	This topic shows how to use Windows Imaging Component (WIC) to create the texture and the view separately.
How to: Use dynamic resources	You create and use dynamic resources when your app needs to change data in those resources. You can create textures and buffers for dynamic usage.
How To: Create a Compute Shader	This topic shows how to create a compute shader.

TOPIC	DESCRIPTION
How To: Design a Hull Shader	This topics shows how to design a hull shader.
How To: Create a Hull Shader	This topic shows how to create a hull shader.
How To: Initialize the Tessellator Stage	This topic shows how to initialize the tessellator stage.
How To: Design a Domain Shader	This topics shows how to design a domain shader.
How To: Create a Domain Shader	This topic shows how to create a domain shader.
How To: Compile a Shader	This topic shows how to use the D3DCompileFromFile function at run time to compile shader code.
How to: Record a Command List	This topic shows how to create and record a command list .
How to: Play Back a Command List	This topic shows how to play back a command list .
How To: Check for Driver Support	This topic shows how to determine whether multithreading features (including resource creation and command lists) are supported for hardware acceleration.

Related topics

[Direct3D 11 Graphics](#)

How To: Create a Reference Device

2/22/2020 • 2 minutes to read • [Edit Online](#)

This topic shows how to create a reference device that implements a highly accurate, software implementation of the runtime. To create a reference device, simply specify that the device you are creating will use a reference driver. This example creates a device and a swap chain at the same time.

To create a reference device

1. Define initial parameters for a swap chain.

```
DXGI_SWAP_CHAIN_DESC sd;
ZeroMemory( &sd, sizeof( sd ) );
sd.BufferCount = 1;
sd.BufferDesc.Width = 640;
sd.BufferDesc.Height = 480;
sd.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
sd.BufferDesc.RefreshRate.Numerator = 60;
sd.BufferDesc.RefreshRate.Denominator = 1;
sd.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
sd.OutputWindow = g_hWnd;
sd.SampleDesc.Count = 1;
sd.SampleDesc.Quality = 0;
sd.Windowed = TRUE;
```

2. Request a feature level that implements the features your application will need. A reference device can be successfully created for the Direct3D 11 runtime.

```
D3D_FEATURE_LEVEL FeatureLevels = D3D_FEATURE_LEVEL_11_0;
```

See more about feature levels in the [D3D_FEATURE_LEVEL](#) enumeration.

3. Create the device by calling [D3D11CreateDeviceAndSwapChain](#).

```
HRESULT hr = S_OK;
D3D_FEATURE_LEVEL FeatureLevel;

if( FAILED( hr = D3D11CreateDeviceAndSwapChain( NULL,
                                              D3D_DRIVER_TYPE_REFERENCE,
                                              NULL,
                                              0,
                                              &FeatureLevel,
                                              1,
                                              D3D11_SDK_VERSION,
                                              &sd,
                                              &g_pSwapChain,
                                              &g_pd3dDevice,
                                              &FeatureLevel,
                                              &g_pImmediateContext ))) {
    return hr;
}
```

You will need to supply the API call with the reference driver type from the [D3D_DRIVER_TYPE](#) enumeration. After the method succeeds, it will return a swap chain interface, a device interface, a pointer to the feature level

that was granted by the driver, and an immediate context interface.

For information about limitations creating a reference device on certain feature levels, see [Limitations Creating WARP and Reference Devices](#).[How to Use Direct3D 11](#)

Related topics

[Devices](#)

[How to Use Direct3D 11](#)

How To: Create a WARP Device

11/2/2020 • 2 minutes to read • [Edit Online](#)

This topic shows how to create a WARP device that implements a high speed software rasterizer. To create a WARP device, simply specify that the device you are creating will use a WARP driver. This example creates a device and a swap chain at the same time.

To create a WARP device

1. Define initial parameters for a swap chain.

```
DXGI_SWAP_CHAIN_DESC sd;
ZeroMemory( &sd, sizeof( sd ) );
sd.BufferCount = 1;
sd.BufferDesc.Width = 640;
sd.BufferDesc.Height = 480;
sd.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
sd.BufferDesc.RefreshRate.Numerator = 60;
sd.BufferDesc.RefreshRate.Denominator = 1;
sd.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
sd.OutputWindow = g_hWnd;
sd.SampleDesc.Count = 1;
sd.SampleDesc.Quality = 0;
sd.Windowed = TRUE;
```

2. Request a feature level that implements the features your application will need. A WARP device can be successfully created for feature levels D3D_FEATURE_LEVEL_9_1 through D3D_FEATURE_LEVEL_10_1 and starting with Windows 8 for all feature levels.

```
D3D_FEATURE_LEVEL FeatureLevels = D3D_FEATURE_LEVEL_10_1;
```

See more about feature levels in the [D3D_FEATURE_LEVEL](#) enumeration.

3. Create the device by calling [D3D11CreateDeviceAndSwapChain](#).

```
HRESULT hr = S_OK;
if( FAILED (hr = D3D11CreateDeviceAndSwapChain( NULL,
                                                D3D_DRIVER_TYPE_WARP,
                                                NULL,
                                                0,
                                                &FeatureLevels,
                                                1,
                                                D3D11_SDK_VERSION,
                                                &sd,
                                                &g_pSwapChain,
                                                &g_pd3dDevice,
                                                &FeatureLevel,
                                                &g_pImmediateContext ))) {
    return hr;
}
```

You will need to supply the API call with the WARP driver type from the [D3D_DRIVER_TYPE](#) enumeration. After the method succeeds, it will return a swap chain interface, a device interface, a pointer to the feature level that was granted by the driver, and an immediate context interface.

For information about limitations creating a WARP device on certain feature levels, see [Limitations Creating WARP and Reference Devices](#).

New for Windows 8

When a computer's primary display adapter is the "Microsoft Basic Display Adapter" (WARP adapter), that computer also has a second adapter. This second adapter is the render-only device that has no display outputs. For more info about the render-only device, see [new info in Windows 8 about enumerating adapters](#).

Related topics

[Devices](#)

[How to Use Direct3D 11](#)

How To: Create a Swap Chain

11/2/2020 • 2 minutes to read • [Edit Online](#)

This topic shows how to create a swap chain that encapsulates two or more buffers that are used for rendering and display. They usually contain a front buffer that is presented to the display device and a back buffer that serves as the render target. After the immediate context is done rendering to the back buffer, the swap chain presents the back buffer by swapping the two buffers.

The swap chain defines several rendering characteristics including:

- The size of the render area.
- The display refresh rate.
- The display mode.
- The surface format.

Define the characteristics of the swap chain by filling in a [DXGI_SWAP_CHAIN_DESC](#) structure and initializing an [IDXGISwapChain](#) interface. Initialize a swap chain by calling [IDXGIFactory::CreateSwapChain](#) or [D3D11CreateDeviceAndSwapChain](#).

Create a device and a swap chain

To initialize a device and swap chain, use one of the following two functions:

- Use the [D3D11CreateDeviceAndSwapChain](#) function when you want to initialize the swap chain at the same time as device initialization. This usually is the easiest option.
- Use the [D3D11CreateDevice](#) function when you have already created a swap chain using [IDXGIFactory::CreateSwapChain](#).

Related topics

[Devices](#)

[How to Use Direct3D 11](#)

How To: Enumerate Adapters

11/2/2020 • 2 minutes to read • [Edit Online](#)

This topic shows how to use Microsoft DirectX Graphics Infrastructure (DXGI) to enumerate the available graphics adapters on a computer. Direct3D 10 and 11 can use DXGI to enumerate adapters.

You generally need to enumerate adapters for these reasons:

- To determine how many graphics adapters are installed on a computer.
- To help you choose which adapter to use to create a Direct3D device.
- To retrieve an [IDXGIAAdapter](#) object that you can use to retrieve device capabilities.

To enumerate adapters

1. Create an [IDXGIFactory](#) object by calling the [CreateDXGIFactory](#) function. The following code example demonstrates how to initialize an [IDXGIFactory](#) object.

```
IDXGIFactory * pFactory = NULL;  
  
CreateDXGIFactory(__uuidof(IDXGIFactory) ,(void**)&pFactory)
```

2. Enumerate through each adapter by calling the [IDXGIFactory::EnumAdapters](#) method. The *Adapter* parameter allows you to specify a zero-based index number of the adapter to enumerate. If no adapter is available at the specified index, the method returns [DXGI_ERROR_NOT_FOUND](#).

The following code example demonstrates how to enumerate through the adapters on a computer.

```
for (UINT i = 0;  
     pFactory->EnumAdapters(i, &pAdapter) != DXGI_ERROR_NOT_FOUND;  
     ++i)  
{ ... }
```

The following code example demonstrates how to enumerate all adapters on a computer.

NOTE

For Direct3D 11.0 and later, we recommend that apps always use [IDXGIFactory1](#) and [CreateDXGIFactory1](#) instead.

```
std::vector<IDXGIAdapter*> EnumerateAdapters(void)
{
    IDXGIAdapter * pAdapter;
    std::vector<IDXGIAdapter*> vAdapters;
    IDXGIFactory* pFactory = NULL;

    // Create a DXGIFactory object.
    if(FAILED(CreateDXGIFactory(__uuidof(IDXGIFactory) ,(void**)&pFactory)))
    {
        return vAdapters;
    }

    for ( UINT i = 0;
          pFactory->EnumAdapters(i, &pAdapter) != DXGI_ERROR_NOT_FOUND;
          ++i )
    {
        vAdapters.push_back(pAdapter);
    }

    if(pFactory)
    {
        pFactory->Release();
    }

    return vAdapters;
}
```

Related topics

[How to Use Direct3D 11](#)

How To: Get Adapter Display Modes

11/2/2020 • 2 minutes to read • [Edit Online](#)

This topic shows how to use Microsoft DirectX Graphics Infrastructure (DXGI) to get the valid display modes associated with an adapter. DirectX 10 and 11 can use DXGI to get the valid display modes. Knowing the valid display modes ensures that your application can properly choose a valid full-screen mode.

To get adapter display modes

1. Create an [IDXGIFactory](#) object and use it to enumerate the available adapters. For more information, see [How To: Enumerate Adapters](#).
2. Call `IDXGIAdapter::EnumOutputs` to enumerate the outputs for each adapter.

```
IDXGIOOutput* pOutput = NULL;  
HRESULT hr;  
  
hr = pAdapter->EnumOutputs(0,&pOutput);
```

3. Call `IDXGIOOutput::GetDisplayModeList` to retrieve an array of [DXGI_MODE_DESC](#) structures and the number of elements in the array. Each `DXGI_MODE_DESC` structure represents a valid display mode for the output.

```
UINT numModes = 0;  
DXGI_MODE_DESC* displayModes = NULL;  
DXGI_FORMAT format = DXGI_FORMAT_R32G32B32A32_FLOAT;  
  
// Get the number of elements  
hr = pOutput->GetDisplayModeList( format, 0, &numModes, NULL );  
  
displayModes = new DXGI_MODE_DESC[numModes];  
  
// Get the list  
hr = pOutput->GetDisplayModeList( format, 0, &numModes, displayModes );
```

Related topics

[Devices](#)

[How to Use Direct3D 11](#)

How To: Create a Device and Immediate Context

11/2/2020 • 2 minutes to read • [Edit Online](#)

This topics shows how to initialize a [device](#). Initializing a [device](#) is one of the first tasks that your application must complete before you can render your scene.

To create a device and immediate context

Fill out the [DXGI_SWAP_CHAIN_DESC](#) structure with information about buffer formats and dimensions. For more information, see [Creating a Swap Chain](#).

The following code example demonstrates how to fill in the [DXGI_SWAP_CHAIN_DESC](#) structure.

```
DXGI_SWAP_CHAIN_DESC sd;
ZeroMemory( &sd, sizeof( sd ) );
sd.BufferCount = 1;
sd.BufferDesc.Width = 640;
sd.BufferDesc.Height = 480;
sd.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
sd.BufferDesc.RefreshRate.Numerator = 60;
sd.BufferDesc.RefreshRate.Denominator = 1;
sd.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
sd.OutputWindow = g_hWnd;
sd.SampleDesc.Count = 1;
sd.SampleDesc.Quality = 0;
sd.Windowed = TRUE;
```

Using the [DXGI_SWAP_CHAIN_DESC](#) structure from step one, call [D3D11CreateDeviceAndSwapChain](#) to initialize the device and swap chain at the same time.

```
D3D_FEATURE_LEVEL FeatureLevelsRequested = D3D_FEATURE_LEVEL_11_0;
UINT numLevelsRequested = 1;
D3D_FEATURE_LEVEL FeatureLevelsSupported;

if( FAILED( hr = D3D11CreateDeviceAndSwapChain( NULL,
                                              D3D_DRIVER_TYPE_HARDWARE,
                                              NULL,
                                              0,
                                              &FeatureLevelsRequested,
                                              numFeatureLevelsRequested,
                                              D3D11_SDK_VERSION,
                                              &sd,
                                              &g_pSwapChain,
                                              &g_pd3dDevice,
                                              &FeatureLevelsSupported,
                                              &g_pImmediateContext )) )
{
    return hr;
}
```

NOTE

If you request a [D3D_FEATURE_LEVEL_11_1](#) device on a computer with only the Direct3D 11.0 runtime, [D3D11CreateDeviceAndSwapChain](#) immediately exits with [E_INVALIDARG](#). To safely request all possible feature levels on a computer with the DirectX 11.0 or DirectX 11.1 runtime, use this code:

```
const D3D_FEATURE_LEVEL lvl[] = { D3D_FEATURE_LEVEL_11_1, D3D_FEATURE_LEVEL_11_0,
D3D_FEATURE_LEVEL_10_1, D3D_FEATURE_LEVEL_10_0,
D3D_FEATURE_LEVEL_9_3, D3D_FEATURE_LEVEL_9_2, D3D_FEATURE_LEVEL_9_1 };
UINT createDeviceFlags = 0;
#ifndef _DEBUG
createDeviceFlags |= D3D11_CREATE_DEVICE_DEBUG;
#endif

ID3D11Device* device = nullptr;
HRESULT hr = D3D11CreateDeviceAndSwapChain( nullptr, D3D_DRIVER_TYPE_HARDWARE, nullptr, createDeviceFlags, lvl, _countof(lvl),
D3D11_SDK_VERSION, &sd, &g_pSwapChain, &g_pd3ddevice, &FeatureLevelsSupported, &g_pImmediateContext );
if ( hr == E_INVALIDARG )
{
    hr = D3D11CreateDeviceAndSwapChain( nullptr, D3D_DRIVER_TYPE_HARDWARE, nullptr, createDeviceFlags, &lvl[1], _countof(lvl) - 1,
D3D11_SDK_VERSION, &sd, &g_pSwapChain, &g_pd3ddevice, &FeatureLevelsSupported, &g_pImmediateContext );
}

if (FAILED(hr))
return hr;
```

Create a render-target view by calling [ID3D11Device::CreateRenderTargetView](#) and bind the back-buffer as a render target by calling [ID3D11DeviceContext::OMSetRenderTargets](#).

```
ID3D11Texture2D* pBackBuffer;
// Get a pointer to the back buffer
hr = g_pSwapChain->GetBuffer( 0, __uuidof( ID3D11Texture2D ),
( LPVOID* )&pBackBuffer );

// Create a render-target view
g_pd3dDevice->CreateRenderTargetView( pBackBuffer, NULL,
&g_pRenderTargetView );

// Bind the view
g_pImmediateContext->OMSetRenderTargets( 1, &g_pRenderTargetView, NULL );
```

Create a viewport to define which parts of the render target will be visible. Define the viewport using the [D3D11_VIEWPORT](#) structure and set the viewport using the [ID3D11DeviceContext::RSSetViewports](#) method.

C++

```
// Setup the viewport
D3D11_VIEWPORT vp;
vp.Width = 640;
vp.Height = 480;
vp.MinDepth = 0.0f;
vp.MaxDepth = 1.0f;
vp.TopLeftX = 0;
vp.TopLeftY = 0;
g_pImmediateContext->RSSetViewports( 1, &vp );
```

Related topics

Devices

How to Use Direct3D 11

> > > > >

How To: Get the Device Feature Level

2/22/2020 • 2 minutes to read • [Edit Online](#)

This topic shows how to get the highest **feature level** supported by a **device**. Direct3D 11 devices support a fixed set of feature levels that are defined in the **D3D_FEATURE_LEVEL** enumeration. When you know the highest **feature level** supported by a device, you can run code paths that are appropriate for that device.

To get the device feature level

1. Call either the **D3D11CreateDevice** function or the **D3D11CreateDeviceAndSwapChain** functions while specifying **NULL** for the *ppDevice* parameter. You can do this before device creation.
 - or -Call **ID3D11Device::GetFeatureLevel** after device creation.
2. Examine the value of the returned **D3D_FEATURE_LEVEL** enumeration from the last step to determine the supported feature level.

The following code example demonstrates how to determine the highest supported feature level by calling the **D3D11CreateDevice** function. **D3D11CreateDevice** stores the highest supported feature level in the **FeatureLevel** variable. You can use this code to examine the value of the **D3D_FEATURE_LEVEL** enumerated type that **D3D11CreateDevice** returns. Note that this code lists all feature levels explicitly (for Direct3D 11.1 and Direct3D 11.2).

NOTE

If the Direct3D 11.1 runtime is present on the computer and *pFeatureLevels* is set to **NULL**, this function won't create a **D3D_FEATURE_LEVEL_11_1** device. To create a **D3D_FEATURE_LEVEL_11_1** device, you must explicitly provide a **D3D_FEATURE_LEVEL** array that includes **D3D_FEATURE_LEVEL_11_1**. If you provide a **D3D_FEATURE_LEVEL** array that contains **D3D_FEATURE_LEVEL_11_1** on a computer that doesn't have the Direct3D 11.1 runtime installed, this function immediately fails with **E_INVALIDARG**.

```
HRESULT hr = E_FAIL;
D3D_FEATURE_LEVEL MaxSupportedFeatureLevel = D3D_FEATURE_LEVEL_9_1;
D3D_FEATURE_LEVEL FeatureLevels[] = {
    D3D_FEATURE_LEVEL_11_1,
    D3D_FEATURE_LEVEL_11_0,
    D3D_FEATURE_LEVEL_10_1,
    D3D_FEATURE_LEVEL_10_0,
    D3D_FEATURE_LEVEL_9_3,
    D3D_FEATURE_LEVEL_9_2,
    D3D_FEATURE_LEVEL_9_1
};

hr = D3D11CreateDevice(
    NULL,
    D3D_DRIVER_TYPE_HARDWARE,
    NULL,
    0,
    &FeatureLevels,
    ARRSIZE(FeatureLevels),
    D3D11_SDK_VERSION,
    NULL,
    &MaxSupportedFeatureLevel,
    NULL
);

if(FAILED(hr))
{
    return hr;
}
```

The [10Level9 Reference](#) section lists the differences between how various [ID3D11Device](#) and [ID3D11DeviceContext](#) methods behave at various 10Level9 feature levels.

Related topics

[Direct3D 11 on Downlevel Hardware](#)

[How to Use Direct3D 11](#)

How to: Create a Vertex Buffer

2/22/2020 • 2 minutes to read • [Edit Online](#)

[Vertex buffers](#) contain per vertex data. This topic shows how to initialize a static [vertex buffer](#), that is, a vertex buffer that does not change. For help initializing a non-static buffer, see the [remarks](#) section.

To initialize a static vertex buffer

1. Define a structure that describes a vertex. For example, if your vertex data contains position data and color data, your structure would have one vector that describes the position and another that describes the color.
2. Allocate memory (using malloc or new) for the structure that you defined in step one. Fill this buffer with the actual vertex data that describes your geometry.
3. Create a buffer description by filling in a [D3D11_BUFFER_DESC](#) structure. Pass the `D3D11_BIND_VERTEX_BUFFER` flag to the `BindFlags` member and pass the size of the structure from step one to the `ByteWidth` member.
4. Create a subresource data description by filling in a [D3D11_SUBRESOURCE_DATA](#) structure. The `pSysMem` member of the [D3D11_SUBRESOURCE_DATA](#) structure should point directly to the resource data created in step two.
5. Call [ID3D11Device::CreateBuffer](#) while passing the [D3D11_BUFFER_DESC](#) structure, the [D3D11_SUBRESOURCE_DATA](#) structure, and the address of a pointer to the [ID3D11Buffer](#) interface to initialize.

The following code example demonstrates how to create a vertex buffer. This example assumes that `g_pd3dDevice` is a valid [ID3D11Device](#) object.

```

ID3D11Buffer* g_pVertexBuffer;

// Define the data-type that
// describes a vertex.
struct SimpleVertexCombined
{
    XMFLOAT3 Pos;
    XMFLOAT3 Col;
};

// Supply the actual vertex data.
SimpleVertexCombined verticesCombo[] =
{
    XMFLOAT3( 0.0f, 0.5f, 0.5f ),
    XMFLOAT3( 0.0f, 0.0f, 0.5f ),
    XMFLOAT3( 0.5f, -0.5f, 0.5f ),
    XMFLOAT3( 0.5f, 0.0f, 0.0f ),
    XMFLOAT3( -0.5f, -0.5f, 0.5f ),
    XMFLOAT3( 0.0f, 0.5f, 0.0f ),
};

// Fill in a buffer description.
D3D11_BUFFER_DESC bufferDesc;
bufferDesc.Usage          = D3D11_USAGE_DEFAULT;
bufferDesc.ByteWidth      = sizeof( SimpleVertexCombined ) * 3;
bufferDesc.BindFlags      = D3D11_BIND_VERTEX_BUFFER;
bufferDesc.CPUAccessFlags = 0;
bufferDesc.MiscFlags      = 0;

// Fill in the subresource data.
D3D11_SUBRESOURCE_DATA InitData;
InitData.pSysMem = verticesCombo;
InitData.SysMemPitch = 0;
InitData.SysMemSlicePitch = 0;

// Create the vertex buffer.
hr = g_pd3dDevice->CreateBuffer( &bufferDesc, &InitData, &g_pVertexBuffer );

```

Remarks

Here are some ways to initialize a vertex buffer that changes over time.

1. Create a 2nd buffer with [D3D11_USAGE_STAGING](#); fill the second buffer using [ID3D11DeviceContext::Map](#), [ID3D11DeviceContext::Unmap](#); use [ID3D11DeviceContext::CopyResource](#) to copy from the staging buffer to the default buffer.
2. Use [ID3D11DeviceContext::UpdateSubresource](#) to copy data from memory.
3. Create a buffer with [D3D11_USAGE_DYNAMIC](#), and fill it with [ID3D11DeviceContext::Map](#), [ID3D11DeviceContext::Unmap](#) (using the Discard and NoOverwrite flags appropriately).

#1 and #2 are useful for content that changes less than once per frame. In general, GPU reads will be fast and CPU updates will be slower.

#3 is useful for content that changes more than once per frame. In general, GPU reads will be slower, but CPU updates will be faster.

Related topics

[Buffers](#)

[How to Use Direct3D 11](#)

How to: Create an Index Buffer

2/22/2020 • 2 minutes to read • [Edit Online](#)

[Index buffers](#) contain index data. This topic shows how to initialize an [index buffer](#) in preparation for rendering.

To initialize an index buffer

1. Create a buffer that contains your index information.
2. Create a buffer description by filling in a [D3D11_BUFFER_DESC](#) structure. Pass the `D3D11_BIND_INDEX_BUFFER` flag to the `BindFlags` member and pass the size of the buffer in bytes to the `ByteWidth` member.
3. Create a subresource data description by filling in a [D3D11_SUBRESOURCE_DATA](#) structure. The `pSysMem` member should point directly to the index data created in step one.
4. Call [ID3D11Device::CreateBuffer](#) while passing the [D3D11_BUFFER_DESC](#) structure, the [D3D11_SUBRESOURCE_DATA](#) structure, and the address of a pointer to the [ID3D11Buffer](#) interface to initialize.

The following code example demonstrates how to create an index buffer. This example assumes that

```
g_pd3dDevice
```

is a valid [ID3D11Device](#) object and that

```
g_pd3dContext
```

is a valid [ID3D11DeviceContext](#) object.

```
ID3D11Buffer *g_pIndexBuffer = NULL;

// Create indices.
unsigned int indices[] = { 0, 1, 2 };

// Fill in a buffer description.
D3D11_BUFFER_DESC bufferDesc;
bufferDesc.Usage      = D3D11_USAGE_DEFAULT;
bufferDesc.ByteWidth   = sizeof( unsigned int ) * 3;
bufferDesc.BindFlags   = D3D11_BIND_INDEX_BUFFER;
bufferDesc.CPUAccessFlags = 0;
bufferDesc.MiscFlags   = 0;

// Define the resource data.
D3D11_SUBRESOURCE_DATA InitData;
InitData.pSysMem = indices;
InitData.SysMemPitch = 0;
InitData.SysMemSlicePitch = 0;

// Create the buffer with the device.
hr = g_pd3dDevice->CreateBuffer( &bufferDesc, &InitData, &g_pIndexBuffer );
if( FAILED( hr ) )
    return hr;

// Set the buffer.
g_pd3dContext->IASetIndexBuffer( g_pIndexBuffer, DXGI_FORMAT_R32_UINT, 0 );
```

Related topics

[Buffers](#)

[How to Use Direct3D 11](#)

How to: Create a Constant Buffer

11/2/2020 • 2 minutes to read • [Edit Online](#)

Constant buffers contain shader constant data. This topic shows how to initialize a [constant buffer](#) in preparation for rendering.

To initialize a constant buffer

1. Define a structure that describes the vertex shader constant data.
2. Allocate memory for the structure that you defined in step one. Fill this buffer with vertex shader constant data. You can use `malloc` or `new` to allocate the memory, or you can allocate memory for the structure from the stack.
3. Create a buffer description by filling in a [D3D11_BUFFER_DESC](#) structure. Pass the `D3D11_BIND_CONSTANT_BUFFER` flag to the `BindFlags` member and pass the size of the constant buffer description structure in bytes to the `ByteWidth` member.

NOTE

The `D3D11_BIND_CONSTANT_BUFFER` flag cannot be combined with any other flags.

4. Create a subresource data description by filling in a [D3D11_SUBRESOURCE_DATA](#) structure. The `pSysMem` member of the [D3D11_SUBRESOURCE_DATA](#) structure must point directly to the vertex shader constant data that you created in step two.
5. Call `ID3D11Device::CreateBuffer` while passing the [D3D11_BUFFER_DESC](#) structure, the [D3D11_SUBRESOURCE_DATA](#) structure, and the address of a pointer to the [ID3D11Buffer](#) interface to initialize.

These code examples demonstrate how to create a constant buffer.

This example assumes that `g_pd3dDevice` is a valid [ID3D11Device](#) object and that `g_pd3dContext` is a valid [ID3D11DeviceContext](#) object.

```

ID3D11Buffer* g_pConstantBuffer11 = NULL;

// Define the constant data used to communicate with shaders.
struct VS_CONSTANT_BUFFER
{
    XMFLOAT4X4 mWorldViewProj;
    XMFLOAT4 vSomeVectorThatMayBeNeededByASpecificShader;
    float fSomeFloatThatMayBeNeededByASpecificShader;
    float fTime;
    float fSomeFloatThatMayBeNeededByASpecificShader2;
    float fSomeFloatThatMayBeNeededByASpecificShader3;
} VS_CONSTANT_BUFFER;

// Supply the vertex shader constant data.
VS_CONSTANT_BUFFER VsConstData;
VsConstData.mWorldViewProj = {...};
VsConstData.vSomeVectorThatMayBeNeededByASpecificShader = XMFLOAT4(1,2,3,4);
VsConstData.fSomeFloatThatMayBeNeededByASpecificShader = 3.0f;
VsConstData.fTime = 1.0f;
VsConstData.fSomeFloatThatMayBeNeededByASpecificShader2 = 2.0f;
VsConstData.fSomeFloatThatMayBeNeededByASpecificShader3 = 4.0f;

// Fill in a buffer description.
D3D11_BUFFER_DESC cbDesc;
cbDesc.ByteWidth = sizeof( VS_CONSTANT_BUFFER );
cbDesc.Usage = D3D11_USAGE_DYNAMIC;
cbDesc.BindFlags = D3D11_BIND_CONSTANT_BUFFER;
cbDesc.CPUAccessFlags = D3D11_CPU_ACCESS_WRITE;
cbDesc.MiscFlags = 0;
cbDesc.StructureByteStride = 0;

// Fill in the subresource data.
D3D11_SUBRESOURCE_DATA InitData;
InitData.pSysMem = &VsConstData;
InitData.SysMemPitch = 0;
InitData.SysMemSlicePitch = 0;

// Create the buffer.
hr = g_pd3dDevice->CreateBuffer( &cbDesc, &InitData,
                                    &g_pConstantBuffer11 );

if( FAILED( hr ) )
    return hr;

// Set the buffer.
g_pd3dContext->VSSetConstantBuffers( 0, 1, &g_pConstantBuffer11 );

```

This example shows the associated HLSL cbuffer definition.

```

cbuffer VS_CONSTANT_BUFFER : register(b0)
{
    matrix mWorldViewProj;
    float4 vSomeVectorThatMayBeNeededByASpecificShader;
    float fSomeFloatThatMayBeNeededByASpecificShader;
    float fTime;
    float fSomeFloatThatMayBeNeededByASpecificShader2;
    float fSomeFloatThatMayBeNeededByASpecificShader3;
};

```

NOTE

Make sure to match the VS_CONSTANT_BUFFER memory layout in C++ with the HLSL layout. For info about how HLSL handles layout in memory, see [Packing Rules for Constant Variables](#).

Related topics

[Buffers](#)

[How to Use Direct3D 11](#)

How to: Create a Texture

2/22/2020 • 2 minutes to read • [Edit Online](#)

The simplest way to create a texture is to describe its properties and call the texture creation API. This topic shows how to create a texture.

To create a texture

1. Fill in a [D3D11_TEXTURE2D_DESC](#) structure with a description of the texture parameters.
2. Create the texture by calling [ID3D11Device::CreateTexture2D](#) with the texture description.

This example creates a 256 x 256 texture, with [dynamic usage](#), for use as a [shader resource](#) with [cpu write access](#).

```
D3D11_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 = D3D11_USAGE_DYNAMIC;
desc.BindFlags = D3D11_BIND_SHADER_RESOURCE;
desc.CPUAccessFlags = D3D11_CPU_ACCESS_WRITE;
desc.MiscFlags = 0;

ID3D11Device *pd3dDevice; // Don't forget to initialize this
ID3D11Texture2D *pTexture = NULL;
pd3dDevice->CreateTexture2D( &desc, NULL, &pTexture );
```

Related topics

[How to Use Direct3D 11](#)

[Textures](#)

How to: Initialize a Texture Programmatically

2/22/2020 • 2 minutes to read • [Edit Online](#)

You can initialize a texture during object creation, or you can fill the object programmatically after it is created. This topic has several examples showing how to initialize textures that are created with different types of usages. This example assumes you already know how to [Create a Texture](#).

- [Default Usage](#)
- [Dynamic Usage](#)
- [Staging Usage](#)
- [Related topics](#)

Default Usage

The most common type of usage is default usage. To fill a default texture (one created with `D3D11_USAGE_DEFAULT`) you can either:

- Call `ID3D11Device::CreateTexture2D` and initialize *pInitialData* to point to data provided from an application.

or
- After calling `ID3D11Device::CreateTexture2D`, use `ID3D11DeviceContext::UpdateSubresource` to fill the default texture with data from a pointer provided by the application.

Dynamic Usage

To fill a dynamic texture (one created with `D3D11_USAGE_DYNAMIC`):

1. Get a pointer to the texture memory by passing in `D3D11_MAP_WRITE_DISCARD` when calling `ID3D11DeviceContext::Map`.
2. Write data to the memory.
3. Call `ID3D11DeviceContext::Unmap` when you are finished writing data.

Staging Usage

To fill a staging texture (one created with `D3D11_USAGE_STAGING`):

1. Get a pointer to the texture memory by passing in `D3D11_MAP_WRITE` when calling `ID3D11DeviceContext::Map`.
2. Write data to the memory.
3. Call `ID3D11DeviceContext::Unmap` when you are finished writing data.

A staging texture can then be used as the source parameter to `ID3D11DeviceContext::CopyResource` or `ID3D11DeviceContext::CopySubresourceRegion` to fill a default or dynamic resource.

Related topics

[How to Use Direct3D 11](#)

[Textures](#)

How to: Initialize a Texture From a File

11/2/2020 • 13 minutes to read • [Edit Online](#)

You can use the [Windows Imaging Component](#) API to initialize a [texture](#) from a file. To load a texture, you must create a texture and a texture view. This topic shows how to use Windows Imaging Component (WIC) to create the texture and the view separately.

NOTE

This topic is useful for images that you create as simple 2D textures. For more complex resources, use [DDS](#). For a full-featured DDS file reader, writer, and texture processing pipeline, see [DirectXTex](#) and [DirectXTK](#).

At the end of this topic, you'll find the full example code. The topic describes the parts of the example code that create the texture and the view.

To initialize a [texture](#) and [view](#) separately.

1. Call [CoCreateInstance](#) to create the imaging factory interface ([IWICImagingFactory](#)).
2. Call the [IWICImagingFactory::CreateDecoderFromFilename](#) method to create a [IWICBitmapDecoder](#) object from an image file name.
3. Call the [IWICBitmapDecoder::GetFrame](#) method to retrieve the [IWICBitmapFrameDecode](#) interface for the frame of the image.
4. Call the [IWICBitmapSource::GetPixelFormat](#) method ([IWICBitmapFrameDecode](#) interface inherits from [IWICBitmapSource](#)) to get the pixel format of the image.
5. Convert the pixel format to a [DXGI_FORMAT](#) type according to this table:

WIC PIXEL FORMAT	EQUIVALENT DXGI_FORMAT
GUID_WICPixelFormat128bppRGBAFloat	DXGI_FORMAT_R32G32B32A32_FLOAT
GUID_WICPixelFormat64bppRGBAHalf	DXGI_FORMAT_R16G16B16A16_FLOAT
GUID_WICPixelFormat64bppRGBA	DXGI_FORMAT_R16G16B16A16_UNORM
GUID_WICPixelFormat32bppRGBA	DXGI_FORMAT_R8G8B8A8_UNORM
GUID_WICPixelFormat32bppBGRA	DXGI_FORMAT_B8G8R8A8_UNORM (DXGI 1.1)
GUID_WICPixelFormat32bppBGR	DXGI_FORMAT_B8G8R8X8_UNORM (DXGI 1.1)
GUID_WICPixelFormat32bppRGBA1010102XR	DXGI_FORMAT_R10G10B10_XR_BIAS_A2_UNORM (DXGI 1.1)
GUID_WICPixelFormat32bppRGBA1010102	DXGI_FORMAT_R10G10B10A2_UNORM
GUID_WICPixelFormat32bppRGBE	DXGI_FORMAT_R9G9B9E5_SHAREDEXP

WIC PIXEL FORMAT	EQUIVALENT DXGI_FORMAT
GUID_WICPixelFormat16bppBGRA5551	DXGI_FORMAT_B5G5R5A1_UNORM (DXGI 1.2)
GUID_WICPixelFormat16bppBGR565	DXGI_FORMAT_B5G6R5_UNORM (DXGI 1.2)
GUID_WICPixelFormat32bppGrayFloat	DXGI_FORMAT_R32_FLOAT*
GUID_WICPixelFormat16bppGrayHalf	DXGI_FORMAT_R16_FLOAT*
GUID_WICPixelFormat16bppGray	DXGI_FORMAT_R16_UNORM*
GUID_WICPixelFormat8bppGray	DXGI_FORMAT_R8_UNORM*
GUID_WICPixelFormat8bppAlpha	DXGI_FORMAT_A8_UNORM
GUID_WICPixelFormat96bppRGBFloat (Windows 8 WIC)	DXGI_FORMAT_R32G32B32_FLOAT

* The single-channel DXGI formats are all red channel, so you need HLSL shader swizzles such as .rrr to render these as grayscale.

6. Call the [IWICBitmapSource::CopyPixels](#) method to copy the image pixels into a buffer. Use the [DXGI_FORMAT](#) type and the buffer to initialize the 2D texture resource and shader-resource-view object.
7. Call the [ID3D11Device::CreateTexture2D](#) method to initialize the 2D texture resource. In this call, pass the address of an [ID3D11Texture2D](#) interface pointer.

```
// Create texture
D3D11_TEXTURE2D_DESC desc;
desc.Width = width;
desc.Height = height;
desc.MipLevels = 1;
desc.ArraySize = 1;
desc.Format = format;
desc.SampleDesc.Count = 1;
desc.SampleDesc.Quality = 0;
desc.Usage = D3D11_USAGE_DEFAULT;
desc.BindFlags = D3D11_BIND_SHADER_RESOURCE;
desc.CPUAccessFlags = 0;
desc.MiscFlags = 0;

D3D11_SUBRESOURCE_DATA initData;
initData.pSysMem = temp.get();
initData.SysMemPitch = static_cast<UINT>( rowPitch );
initData.SysMemSlicePitch = static_cast<UINT>( imageSize );

ID3D11Texture2D* tex = nullptr;
hr = d3dDevice->CreateTexture2D( &desc, &initData, &tex );
```

8. Call the [ID3D11Device::CreateShaderResourceView](#) method to initialize a shader-resource-view object. Pass either a **NULL** shader-resource-view description (to get a view with default parameters) or a non-**NULL** shader-resource-view description (to get a view with non-default parameters). If necessary, determine the texture type by calling [ID3D11Resource::GetType](#) and the texture format by calling [ID3D11ShaderResourceView::GetDesc](#).

```
if ( SUCCEEDED(hr) && tex != 0 )
{
    if (textureView != 0)
    {
        D3D11_SHADER_RESOURCE_VIEW_DESC SRVDesc;
        memset( &SRVDesc, 0, sizeof( SRVDesc ) );
        SRVDesc.Format = format;
        SRVDesc.ViewDimension = D3D11_SRV_DIMENSION_TEXTURE2D;
        SRVDesc.Texture2D.MipLevels = 1;

        hr = d3dDevice->CreateShaderResourceView( tex, &SRVDesc, textureView );
        if ( FAILED(hr) )
        {
            tex->Release();
            return hr;
        }
    }
}
```

The preceding example code assumes that the *d3dDevice* variable is an [ID3D11Device](#) object that has been previously initialized.

Here is the header that you can include in your app. The header declares the [CreateWICTextureFromFile](#) and [CreateWICTextureFromMemory](#) functions that you can call in your app to create a texture from a file and from memory.

```

//-----
// File: WICTextureLoader.h
//
// Function for loading a WIC image and creating a Direct3D 11 runtime texture for it
// (auto-generating mipmaps if possible)
//
// Note: Assumes application has already called CoInitializeEx
//
// Warning: CreateWICTexture* functions are not thread-safe if given a d3dContext instance for
//           auto-gen mipmap support.
//
// Note these functions are useful for images created as simple 2D textures. For
// more complex resources, DDSTextureLoader is an excellent light-weight runtime loader.
// For a full-featured DDS file reader, writer, and texture processing pipeline see
// the 'Texconv' sample and the 'DirectXTex' library.
//
// THIS CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF
// ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO
// THE IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A
// PARTICULAR PURPOSE.
//
// Copyright (c) Microsoft Corporation. All rights reserved.
//
// https://go.microsoft.com/fwlink/?LinkId=248926
// https://go.microsoft.com/fwlink/?LinkId=248929
//-----

#ifndef _MSC_VER
#pragma once
#endif

#include <d3d11.h>

#pragma warning(push)
#pragma warning(disable : 4005)
#include <stdint.h>
#pragma warning(pop)

HRESULT CreateWICTextureFromMemory( _In_ ID3D11Device* d3dDevice,
                                    _In_opt_ ID3D11DeviceContext* d3dContext,
                                    _In_bytecount_(wicDataSize) const uint8_t* wicData,
                                    _In_size_t wicDataSize,
                                    _Out_opt_ ID3D11Resource** texture,
                                    _Out_opt_ ID3D11ShaderResourceView** textureView,
                                    _In_size_t maxsize = 0
                                );

HRESULT CreateWICTextureFromFile( _In_ ID3D11Device* d3dDevice,
                                 _In_opt_ ID3D11DeviceContext* d3dContext,
                                 _In_z_ const wchar_t* szFileName,
                                 _Out_opt_ ID3D11Resource** texture,
                                 _Out_opt_ ID3D11ShaderResourceView** textureView,
                                 _In_size_t maxsize = 0
                                );

```

Here is the full source that you can use in your app. The source implements the **CreateWICTextureFromFile** and **CreateWICTextureFromMemory** functions.

```

//-----
// File: WICTextureLoader.cpp
//
// Function for loading a WIC image and creating a Direct3D 11 runtime texture for it
// (auto-generating mipmaps if possible)
//
// Note: Assumes application has already called CoInitializeEx
//
```

```


// Warning: CreateWICTexture* functions are not thread-safe if given a d3dContext instance for
//           auto-gen mipmap support.
//
// Note these functions are useful for images created as simple 2D textures. For
// more complex resources, DDSTextureLoader is an excellent light-weight runtime loader.
// For a full-featured DDS file reader, writer, and texture processing pipeline see
// the 'Texconv' sample and the 'DirectXTex' library.
//
// THIS CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF
// ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO
// THE IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A
// PARTICULAR PURPOSE.
//
// Copyright (c) Microsoft Corporation. All rights reserved.
//
// https://go.microsoft.com/fwlink/?LinkId=248926
// https://go.microsoft.com/fwlink/?LinkId=248929
//-----

// We could load multi-frame images (TIFF/GIF) into a texture array.
// For now, we just load the first frame (note: DirectXTex supports multi-frame images)

#include <dxgiformat.h>
#include <assert.h>

#pragma warning(push)
#pragma warning(disable : 4005)
#include <wincodec.h>
#pragma warning(pop)

#include <memory>

#include "WICTextureLoader.h"

#if (_WIN32_WINNT >= 0x0602 /*_WIN32_WINNT_WIN8*/) && !defined(DXGI_1_2_FORMATS)
#define DXGI_1_2_FORMATS
#endif

//-----
template<class T> class ScopedObject
{
public:
    explicit ScopedObject( T *p = 0 ) : _pointer(p) {}
    ~ScopedObject()
    {
        if ( _pointer )
        {
            _pointer->Release();
            _pointer = nullptr;
        }
    }

    bool IsNull() const { return (!_pointer); }

    T& operator*() { return *_pointer; }
    T* operator->() { return _pointer; }
    T** operator&() { return &_pointer; }

    void Reset(T *p = 0) { if ( _pointer ) { _pointer->Release(); } _pointer = p; }

    T* Get() const { return _pointer; }

private:
    ScopedObject(const ScopedObject&);
    ScopedObject& operator=(const ScopedObject&);

    T* _pointer;
};


```

```

//-----
// WIC Pixel Format Translation Data
//-----

struct WICTranslate
{
    GUID           wic;
    DXGI_FORMAT    format;
};

static WICTranslate g_WICFormats[] =
{
    { GUID_WICPixelFormat128bppRGBAFloat,           DXGI_FORMAT_R32G32B32A32_FLOAT },
    { GUID_WICPixelFormat64bppRGBAHalf,              DXGI_FORMAT_R16G16B16A16_FLOAT },
    { GUID_WICPixelFormat64bppRGBA,                  DXGI_FORMAT_R16G16B16A16_UNORM },
    { GUID_WICPixelFormat32bppRGBA,                  DXGI_FORMAT_R8G8B8A8_UNORM },
    { GUID_WICPixelFormat32bppBGRA,                 DXGI_FORMAT_B8G8R8A8_UNORM }, // DXGI 1.1
    { GUID_WICPixelFormat32bppBGR,                   DXGI_FORMAT_B8G8R8X8_UNORM }, // DXGI 1.1
    { GUID_WICPixelFormat32bppRGBA1010102XR,        DXGI_FORMAT_R10G10B10_XR_BIAS_A2_UNORM }, // DXGI 1.1
    { GUID_WICPixelFormat32bppRGBAA1010102,          DXGI_FORMAT_R10G10B10A2_UNORM },
    { GUID_WICPixelFormat32bppRGBE,                  DXGI_FORMAT_R9G9B9E5_SHAREDEXP },
    { GUID_WICPixelFormat16bppBGRA5551,              DXGI_FORMAT_B5G5R5A1_UNORM },
    { GUID_WICPixelFormat16bppBGR565,                DXGI_FORMAT_B5G6R5_UNORM },
};

#endif // DXGI_1_2_FORMATS

#ifndef _WIN32_WINNT
{ GUID_WICPixelFormat32bppGrayFloat,             DXGI_FORMAT_R32_FLOAT },
{ GUID_WICPixelFormat16bppGrayHalf,              DXGI_FORMAT_R16_FLOAT },
{ GUID_WICPixelFormat16bppGray,                  DXGI_FORMAT_R16_UNORM },
{ GUID_WICPixelFormat8bppGray,                  DXGI_FORMAT_R8_UNORM },
{ GUID_WICPixelFormat8bppAlpha,                 DXGI_FORMAT_A8_UNORM },
{ GUID_WICPixelFormat96bppRGBFloat,             DXGI_FORMAT_R32G32B32_FLOAT },
#endif
};

//-----
// WIC Pixel Format nearest conversion table
//-----

struct WICConvert
{
    GUID source;
    GUID target;
};

static WICConvert g_WICConvert[] =
{
    // Note target GUID in this conversion table must be one of those directly supported formats (above).

    { GUID_WICPixelFormatBlackWhite,               GUID_WICPixelFormat8bppGray }, // DXGI_FORMAT_R8_UNORM
    { GUID_WICPixelFormat1bppIndexed,             DXGI_FORMAT_R8G8B8A8_UNORM,   GUID_WICPixelFormat32bppRGBA }, // /
    { GUID_WICPixelFormat2bppIndexed,             DXGI_FORMAT_R8G8B8A8_UNORM,   GUID_WICPixelFormat32bppRGBAA1010102 }, // /
    { GUID_WICPixelFormat4bppIndexed,             DXGI_FORMAT_R8G8B8A8_UNORM,   GUID_WICPixelFormat32bppRGBE }, // /
    { GUID_WICPixelFormat8bppIndexed,             DXGI_FORMAT_R8G8B8A8_UNORM,   GUID_WICPixelFormat32bppRGBAFloat }, // /
    { GUID_WICPixelFormat128bppRGBAFloat,        DXGI_FORMAT_R32G32B32A32_FLOAT }
};

```

```

    { GUID_WICPixelFormat2bppGray,           GUID_WICPixelFormat8bppGray }, // DXGI_FORMAT_R8_UNORM
    { GUID_WICPixelFormat4bppGray,           GUID_WICPixelFormat8bppGray }, // DXGI_FORMAT_R8_UNORM

    { GUID_WICPixelFormat16bppGrayFixedPoint, GUID_WICPixelFormat16bppGrayHalf }, // DXGI_FORMAT_R16_FLOAT
    { GUID_WICPixelFormat32bppGrayFixedPoint, GUID_WICPixelFormat32bppGrayFloat }, // DXGI_FORMAT_R32_FLOAT

#define DXGI_1_2_FORMATS

    { GUID_WICPixelFormat16bppBGR555,         GUID_WICPixelFormat16bppBGRA5551 }, //
DXGI_FORMAT_B5G5R5A1_UNORM

#else

    { GUID_WICPixelFormat16bppBGR555,         GUID_WICPixelFormat32bppRGBA }, //
DXGI_FORMAT_R8G8B8A8_UNORM
    { GUID_WICPixelFormat16bppBGRA5551,        GUID_WICPixelFormat32bppRGBA }, //
DXGI_FORMAT_R8G8B8A8_UNORM
    { GUID_WICPixelFormat16bppBGR565,         GUID_WICPixelFormat32bppRGBA }, //
DXGI_FORMAT_R8G8B8A8_UNORM

#endif // DXGI_1_2_FORMATS

    { GUID_WICPixelFormat32bppBGR101010,       GUID_WICPixelFormat32bppRGBA1010102 }, //
DXGI_FORMAT_R10G10B10A2_UNORM

    { GUID_WICPixelFormat24bppBGR,            GUID_WICPixelFormat32bppRGBA }, //
DXGI_FORMAT_R8G8B8A8_UNORM
    { GUID_WICPixelFormat24bppRGB,            GUID_WICPixelFormat32bppRGBA }, //
DXGI_FORMAT_R8G8B8A8_UNORM
    { GUID_WICPixelFormat32bppPBGRA,          GUID_WICPixelFormat32bppRGBA }, //
DXGI_FORMAT_R8G8B8A8_UNORM
    { GUID_WICPixelFormat32bppPRGBA,          GUID_WICPixelFormat32bppRGBA }, //
DXGI_FORMAT_R8G8B8A8_UNORM

    { GUID_WICPixelFormat48bppRGB,            GUID_WICPixelFormat64bppRGBA }, //
DXGI_FORMAT_R16G16B16A16_UNORM
    { GUID_WICPixelFormat48bppBGR,            GUID_WICPixelFormat64bppRGBA }, //
DXGI_FORMAT_R16G16B16A16_UNORM
    { GUID_WICPixelFormat64bppBGRA,          GUID_WICPixelFormat64bppRGBA }, //
DXGI_FORMAT_R16G16B16A16_UNORM
    { GUID_WICPixelFormat64bppPRGBA,          GUID_WICPixelFormat64bppRGBA }, //
DXGI_FORMAT_R16G16B16A16_UNORM
    { GUID_WICPixelFormat64bppPBGRA,          GUID_WICPixelFormat64bppRGBA }, //
DXGI_FORMAT_R16G16B16A16_UNORM

    { GUID_WICPixelFormat48bppRGBFixedPoint, GUID_WICPixelFormat64bppRGBAHalf }, //
DXGI_FORMAT_R16G16B16A16_FLOAT
    { GUID_WICPixelFormat48bppBGRFixedPoint, GUID_WICPixelFormat64bppRGBAHalf }, //
DXGI_FORMAT_R16G16B16A16_FLOAT
    { GUID_WICPixelFormat64bppRGBAFixedPoint, GUID_WICPixelFormat64bppRGBAHalf }, //
DXGI_FORMAT_R16G16B16A16_FLOAT
    { GUID_WICPixelFormat64bppBGRAFixedPoint, GUID_WICPixelFormat64bppRGBAHalf }, //
DXGI_FORMAT_R16G16B16A16_FLOAT
    { GUID_WICPixelFormat64bppRGBAFixedPoint, GUID_WICPixelFormat64bppRGBAHalf }, //
DXGI_FORMAT_R16G16B16A16_FLOAT
    { GUID_WICPixelFormat64bppRGBHalf,          GUID_WICPixelFormat64bppRGBAHalf }, //
DXGI_FORMAT_R16G16B16A16_FLOAT
    { GUID_WICPixelFormat48bppRGBHalf,          GUID_WICPixelFormat64bppRGBAHalf }, //
DXGI_FORMAT_R16G16B16A16_FLOAT

    { GUID_WICPixelFormat96bppRGBFixedPoint,   GUID_WICPixelFormat128bppRGBAFloat }, //
DXGI_FORMAT_R32G32B32A32_FLOAT
    { GUID_WICPixelFormat128bppPRGBAFloat,     GUID_WICPixelFormat128bppRGBAFloat }, //
DXGI_FORMAT_R32G32B32A32_FLOAT
    { GUID_WICPixelFormat128bppRGBFloat,       GUID_WICPixelFormat128bppRGBAFloat }, //
DXGI_FORMAT_R32G32B32A32_FLOAT
    { GUID_WICPixelFormat128bppRGBAFixedPoint, GUID_WICPixelFormat128bppRGBAFloat }, //
DXGI_FORMAT_R32G32B32A32_FLOAT

```

```

    { GUID_WICPixelFormat128bppRGBFixedPoint,   GUID_WICPixelFormat128bppRGBAFloat }, //
DXGI_FORMAT_R32G32B32A32_FLOAT

    { GUID_WICPixelFormat32bppCMYK,               GUID_WICPixelFormat32bppRGBA }, //
DXGI_FORMAT_R8G8B8A8_UNORM
    { GUID_WICPixelFormat64bppCMYK,               GUID_WICPixelFormat64bppRGBA }, //
DXGI_FORMAT_R16G16B16A16_UNORM
    { GUID_WICPixelFormat40bppCMYKAlpha,         GUID_WICPixelFormat64bppRGBA }, //
DXGI_FORMAT_R16G16B16A16_UNORM
    { GUID_WICPixelFormat80bppCMYKAlpha,         GUID_WICPixelFormat64bppRGBA }, //
DXGI_FORMAT_R16G16B16A16_UNORM

#ifndef (_WIN32_WINNT >= 0x0602 /*_WIN32_WINNT_WIN8*/)
    { GUID_WICPixelFormat32bppRGB,               GUID_WICPixelFormat32bppRGBA }, //
DXGI_FORMAT_R8G8B8A8_UNORM
    { GUID_WICPixelFormat64bppRGB,               GUID_WICPixelFormat64bppRGBA }, //
DXGI_FORMAT_R16G16B16A16_UNORM
    { GUID_WICPixelFormat64bppPRGBAHalf,         GUID_WICPixelFormat64bppPRGBAHalf }, //
DXGI_FORMAT_R16G16B16A16_FLOAT
#endif

// We don't support n-channel formats
};

//-----
static IWICImagingFactory* _GetWIC()
{
    static IWICImagingFactory* s_Factory = nullptr;

    if ( s_Factory )
        return s_Factory;

    HRESULT hr = CoCreateInstance(
        CLSID_WICImagingFactory,
        nullptr,
        CLSCTX_INPROC_SERVER,
        __uuidof(IWICImagingFactory),
        (LPVOID*)&s_Factory
    );

    if ( FAILED(hr) )
    {
        s_Factory = nullptr;
        return nullptr;
    }

    return s_Factory;
}

//-----
static DXGI_FORMAT _WICToDXGI( const GUID& guid )
{
    for( size_t i=0; i < _countof(g_WICFormats); ++i )
    {
        if ( memcmp( &g_WICFormats[i].wic, &guid, sizeof(GUID) ) == 0 )
            return g_WICFormats[i].format;
    }

    return DXGI_FORMAT_UNKNOWN;
}

//-----
static size_t _WICBitsPerPixel( REFGUID targetGuid )
{
    IWICImagingFactory* pWIC = _GetWIC();
    if ( !pWIC )
        return 0;

    ScopedObject<IWICComponentInfo> cinfo;

```

```

    if ( FAILED( pWIC->CreateComponentInfo( targetGuid, &cinfo ) ) )
        return 0;

    WICComponentType type;
    if ( FAILED( cinfo->GetComponentType( &type ) ) )
        return 0;

    if ( type != WICPixelFormat )
        return 0;

    ScopedObject<IWICPixelFormatInfo> pfinfo;
    if ( FAILED( cinfo->QueryInterface( __uuidof(IWICPixelFormatInfo), reinterpret_cast<void**>( &pfinfo ) ) ) )
        return 0;

    UINT bpp;
    if ( FAILED( pfinfo->GetBitsPerPixel( &bpp ) ) )
        return 0;

    return bpp;
}

//-----
static HRESULT CreateTextureFromWIC( _In_ ID3D11Device* d3dDevice,
                                    _In_opt_ ID3D11DeviceContext* d3dContext,
                                    _In_ IWICBitmapFrameDecode *frame,
                                    _Out_opt_ ID3D11Resource** texture,
                                    _Out_opt_ ID3D11ShaderResourceView** textureView,
                                    _In_ size_t maxsize )
{
    UINT width, height;
    HRESULT hr = frame->GetSize( &width, &height );
    if ( FAILED(hr) )
        return hr;

    assert( width > 0 && height > 0 );

    if ( !maxsize )
    {
        // This is a bit conservative because the hardware could support larger textures than
        // the Feature Level defined minimums, but doing it this way is much easier and more
        // performant for WIC than the 'fail and retry' model used by DDSTextureLoader

        switch( d3dDevice->GetFeatureLevel() )
        {
            case D3D_FEATURE_LEVEL_9_1:
            case D3D_FEATURE_LEVEL_9_2:
                maxsize = 2048 /*D3D_FL9_1_REQ_TEXTURE2D_U_OR_V_DIMENSION*/;
                break;

            case D3D_FEATURE_LEVEL_9_3:
                maxsize = 4096 /*D3D_FL9_3_REQ_TEXTURE2D_U_OR_V_DIMENSION*/;
                break;

            case D3D_FEATURE_LEVEL_10_0:
            case D3D_FEATURE_LEVEL_10_1:
                maxsize = 8192 /*D3D10_REQ_TEXTURE2D_U_OR_V_DIMENSION*/;
                break;

            default:
                maxsize = D3D11_REQ_TEXTURE2D_U_OR_V_DIMENSION;
                break;
        }
    }

    assert( maxsize > 0 );

    UINT twidth, theight;
    if ( width > maxsize || height > maxsize )

```

```

{
    float ar = static_cast<float>(height) / static_cast<float>(width);
    if ( width > height )
    {
        twidth = static_cast<UINT>( maxsize );
        theight = static_cast<UINT>( static_cast<float>(maxsize) * ar );
    }
    else
    {
        theight = static_cast<UINT>( maxsize );
        twidth = static_cast<UINT>( static_cast<float>(maxsize) / ar );
    }
    assert( twidth <= maxsize && theight <= maxsize );
}
else
{
    twidth = width;
    theight = height;
}

// Determine format
WICPixelFormatGUID pixelFormat;
hr = frame->GetPixelFormat( &pixelFormat );
if ( FAILED(hr) )
    return hr;

WICPixelFormatGUID convertGUID;
memcpy( &convertGUID, &pixelFormat, sizeof(WICPixelFormatGUID) );

size_t bpp = 0;

DXGI_FORMAT format = _WICToDXGI( pixelFormat );
if ( format == DXGI_FORMAT_UNKNOWN )
{
    for( size_t i=0; i < _countof(g_WICConvert); ++i )
    {
        if ( memcmp( &g_WICConvert[i].source, &pixelFormat, sizeof(WICPixelFormatGUID) ) == 0 )
        {
            memcpy( &convertGUID, &g_WICConvert[i].target, sizeof(WICPixelFormatGUID) );

            format = _WICToDXGI( g_WICConvert[i].target );
            assert( format != DXGI_FORMAT_UNKNOWN );
            bpp = _WICBitsPerPixel( convertGUID );
            break;
        }
    }
}

if ( format == DXGI_FORMAT_UNKNOWN )
    return HRESULT_FROM_WIN32( ERROR_NOT_SUPPORTED );
}
else
{
    bpp = _WICBitsPerPixel( pixelFormat );
}

if ( !bpp )
    return E_FAIL;

// Verify our target format is supported by the current device
// (handles WDDM 1.0 or WDDM 1.1 device driver cases as well as DirectX 11.0 Runtime without 16bpp
format support)
UINT support = 0;
hr = d3dDevice->CheckFormatSupport( format, &support );
if ( FAILED(hr) || !(support & D3D11_FORMAT_SUPPORT_TEXTURE2D) )
{
    // Fallback to RGBA 32-bit format which is supported by all devices
    memcpy( &convertGUID, &GUID_WICPixelFormat32bppRGBA, sizeof(WICPixelFormatGUID) );
    format = DXGI_FORMAT_R8G8B8A8_UNORM;
    bpp = 32;
}

```

```

        }

        // Allocate temporary memory for image
        size_t rowPitch = ( twidth * bpp + 7 ) / 8;
        size_t imageSize = rowPitch * theight;

        std::unique_ptr<uint8_t[]> temp( new uint8_t[ imageSize ] );

        // Load image data
        if ( memcmp( &convertGUID, &pixelFormat, sizeof(GUID) ) == 0
            && twidth == width
            && theight == height )
        {
            // No format conversion or resize needed
            hr = frame->CopyPixels( 0, static_cast<UINT>( rowPitch ), static_cast<UINT>( imageSize ), temp.get()
        );
            if ( FAILED(hr) )
                return hr;
        }
        else if ( twidth != width || theight != height )
        {
            // Resize
            IWICImagingFactory* pWIC = _GetWIC();
            if ( !pWIC )
                return E_NOINTERFACE;

            ScopedObject<IWICBitmapScaler> scaler;
            hr = pWIC->CreateBitmapScaler( &scaler );
            if ( FAILED(hr) )
                return hr;

            hr = scaler->Initialize( frame, twidth, theight, WICBitmapInterpolationModeFant );
            if ( FAILED(hr) )
                return hr;

            WICPixelFormatGUID pfScaler;
            hr = scaler->GetPixelFormat( &pfScaler );
            if ( FAILED(hr) )
                return hr;

            if ( memcmp( &convertGUID, &pfScaler, sizeof(GUID) ) == 0 )
            {
                // No format conversion needed
                hr = scaler->CopyPixels( 0, static_cast<UINT>( rowPitch ), static_cast<UINT>( imageSize ),
temp.get() );
                if ( FAILED(hr) )
                    return hr;
            }
            else
            {
                ScopedObject<IWICFormatConverter> FC;
                hr = pWIC->CreateFormatConverter( &FC );
                if ( FAILED(hr) )
                    return hr;

                hr = FC->Initialize( scaler.Get(), convertGUID, WICBitmapDitherTypeErrorDiffusion, 0, 0,
WICBitmapPaletteTypeCustom );
                if ( FAILED(hr) )
                    return hr;

                hr = FC->CopyPixels( 0, static_cast<UINT>( rowPitch ), static_cast<UINT>( imageSize ),
temp.get() );
                if ( FAILED(hr) )
                    return hr;
            }
        }
        else
        {
            // Format conversion but no resize

```

```

// Format conversion but no resize
IWICImagingFactory* pWIC = _GetWIC();
if ( !pWIC )
    return E_NOINTERFACE;

ScopedObject<IWICFormatConverter> FC;
hr = pWIC->CreateFormatConverter( &FC );
if ( FAILED(hr) )
    return hr;

hr = FC->Initialize( frame, convertGUID, WICBitmapDitherTypeErrorDiffusion, 0, 0,
WICBitmapPaletteTypeCustom );
if ( FAILED(hr) )
    return hr;

hr = FC->CopyPixels( 0, static_cast<UINT>( rowPitch ), static_cast<UINT>( imageSize ), temp.get() );
if ( FAILED(hr) )
    return hr;
}

// See if format is supported for auto-gen mipmaps (varies by feature level)
bool autogen = false;
if ( d3dContext != 0 && textureView != 0 ) // Must have context and shader-view to auto generate mipmaps
{
    UINT fmtSupport = 0;
    hr = d3dDevice->CheckFormatSupport( format, &fmtSupport );
    if ( SUCCEEDED(hr) && ( fmtSupport & D3D11_FORMAT_SUPPORT_MIP_AUTOGEN ) )
    {
        autogen = true;
    }
}

// Create texture
D3D11_TEXTURE2D_DESC desc;
desc.Width = twidth;
desc.Height = theight;
desc.MipLevels = (autogen) ? 0 : 1;
desc.ArraySize = 1;
desc.Format = format;
desc.SampleDesc.Count = 1;
desc.SampleDesc.Quality = 0;
desc.Usage = D3D11_USAGE_DEFAULT;
desc.BindFlags = (autogen) ? (D3D11_BIND_SHADER_RESOURCE | D3D11_BIND_RENDER_TARGET) :
(D3D11_BIND_SHADER_RESOURCE);
desc.CPUAccessFlags = 0;
desc.MiscFlags = (autogen) ? D3D11_RESOURCE_MISC_GENERATE_MIPS : 0;

D3D11_SUBRESOURCE_DATA initData;
initData.pSysMem = temp.get();
initData.SysMemPitch = static_cast<UINT>( rowPitch );
initData.SysMemSlicePitch = static_cast<UINT>( imageSize );

ID3D11Texture2D* tex = nullptr;
hr = d3dDevice->CreateTexture2D( &desc, (autogen) ? nullptr : &initData, &tex );
if ( SUCCEEDED(hr) && tex != 0 )
{
    if ( textureView != 0 )
    {
        D3D11_SHADER_RESOURCE_VIEW_DESC SRVDesc;
        memset( &SRVDesc, 0, sizeof( SRVDesc ) );
        SRVDesc.Format = format;
        SRVDesc.ViewDimension = D3D11_SRV_DIMENSION_TEXTURE2D;
        SRVDesc.Texture2D.MipLevels = (autogen) ? -1 : 1;

        hr = d3dDevice->CreateShaderResourceView( tex, &SRVDesc, textureView );
        if ( FAILED(hr) )
        {
            tex->Release();
            return hr;
        }
    }
}

```

```

        }

        if ( autogen )
        {
            assert( d3dContext != 0 );
            d3dContext->UpdateSubresource( tex, 0, nullptr, temp.get(), static_cast<UINT>(rowPitch),
static_cast<UINT>(imageSize) );
            d3dContext->GenerateMips( *textureView );
        }
    }

    if (texture != 0)
    {
        *texture = tex;
    }
    else
    {
#ifndef _DEBUG || defined(PROFILE)
        tex->SetPrivateData( WKPID_D3DDebugObjectName,
                           sizeof("WICTextureLoader")-1,
                           "WICTextureLoader"
                           );
#endif
        tex->Release();
    }
}

return hr;
}

//-----
HRESULT CreateWICTextureFromMemory( _In_ ID3D11Device* d3dDevice,
                                    _In_opt_ ID3D11DeviceContext* d3dContext,
                                    _In_bytecount_(wicDataSize) const uint8_t* wicData,
                                    _In_ size_t wicDataSize,
                                    _Out_opt_ ID3D11Resource** texture,
                                    _Out_opt_ ID3D11ShaderResourceView** textureView,
                                    _In_ size_t maxsize
                                    )
{
    if (!d3dDevice || !wicData || (!texture && !textureView))
    {
        return E_INVALIDARG;
    }

    if ( !wicDataSize )
    {
        return E_FAIL;
    }

#ifndef _M_AMD64
    if ( wicDataSize > 0xFFFFFFFF )
        return HRESULT_FROM_WIN32( ERROR_FILE_TOO_LARGE );
#endif

    IWICImagingFactory* pWIC = _GetWIC();
    if ( !pWIC )
        return E_NOINTERFACE;

    // Create input stream for memory
    ScopedObject<IWICStream> stream;
    HRESULT hr = pWIC->CreateStream( &stream );
    if ( FAILED(hr) )
        return hr;

    hr = stream->InitializeFromMemory( const_cast<uint8_t*>( wicData ), static_cast<DWORD>( wicDataSize ) );
    if ( FAILED(hr) )
        return hr;
}

```

```

// Initialize WIC
ScopedObject<IWICBitmapDecoder> decoder;
hr = pWIC->CreateDecoderFromStream( stream.Get(), 0, WICDecodeMetadataCacheOnDemand, &decoder );
if ( FAILED(hr) )
    return hr;

ScopedObject<IWICBitmapFrameDecode> frame;
hr = decoder->GetFrame( 0, &frame );
if ( FAILED(hr) )
    return hr;

hr = CreateTextureFromWIC( d3dDevice, d3dContext, frame.Get(), texture, textureView, maxsize );
if ( FAILED(hr) )
    return hr;

#ifndef _DEBUG
if (texture != 0 && *texture != 0)
{
    (*texture)->SetPrivateData( WKPID_D3DDescriptorName,
                                sizeof("WICTextureLoader")-1,
                                "WICTextureLoader"
                                );
}

if (textureView != 0 && *textureView != 0)
{
    (*textureView)->SetPrivateData( WKPID_D3DDescriptorName,
                                    sizeof("WICTextureLoader")-1,
                                    "WICTextureLoader"
                                    );
}
#endif

return hr;
}

//-----
HRESULT CreateWICTextureFromFile( _In_ ID3D11Device* d3dDevice,
                                 _In_opt_ ID3D11DeviceContext* d3dContext,
                                 _In_z_ const wchar_t* fileName,
                                 _Out_opt_ ID3D11Resource** texture,
                                 _Out_opt_ ID3D11ShaderResourceView** textureView,
                                 _In_ size_t maxsize )
{
    if (!d3dDevice || !fileName || (!texture && !textureView))
    {
        return E_INVALIDARG;
    }

    IWICImagingFactory* pWIC = _GetWIC();
    if ( !pWIC )
        return E_NOINTERFACE;

    // Initialize WIC
    ScopedObject<IWICBitmapDecoder> decoder;
    HRESULT hr = pWIC->CreateDecoderFromFilename( fileName, 0, GENERIC_READ, WICDecodeMetadataCacheOnDemand,
&decoder );
    if ( FAILED(hr) )
        return hr;

    ScopedObject<IWICBitmapFrameDecode> frame;
    hr = decoder->GetFrame( 0, &frame );
    if ( FAILED(hr) )
        return hr;

    hr = CreateTextureFromWIC( d3dDevice, d3dContext, frame.Get(), texture, textureView, maxsize );
    if ( FAILED(hr) )
        return hr;
}

```

```

#ifndef _DEBUG || defined(PROFILE)
    if (texture != 0 || textureView != 0)
    {
        CHAR strFileA[MAX_PATH];
        WideCharToMultiByte( CP_ACP,
            WC_NO_BEST_FIT_CHARS,
            fileName,
            -1,
            strFileA,
            MAX_PATH,
            nullptr,
            FALSE
        );
        const CHAR* pstrName = strrchr( strFileA, '\\');
        if (!pstrName)
        {
            pstrName = strFileA;
        }
        else
        {
            pstrName++;
        }

        if (texture != 0 && *texture != 0)
        {
            (*texture)->SetPrivateData( WKPID_D3DDescribeObjectName,
                static_cast<UINT>( strlen_s(pstrName, MAX_PATH) ),
                pstrName
            );
        }

        if (textureView != 0 && *textureView != 0 )
        {
            (*textureView)->SetPrivateData( WKPID_D3DDescribeObjectName,
                static_cast<UINT>( strlen_s(pstrName, MAX_PATH) ),
                pstrName
            );
        }
    }
#endif

    return hr;
}

```

Related topics

[How to Use Direct3D 11](#)

[Textures](#)

How to: Use dynamic resources

2/22/2020 • 6 minutes to read • [Edit Online](#)

Important APIs

- [ID3D11DeviceContext::Map](#)
- [ID3D11DeviceContext::Unmap](#)
- [D3D11_USAGE](#)

You create and use dynamic resources when your app needs to change data in those resources. You can create textures and buffers for dynamic usage.

What you need to know

Technologies

- [How to: Create a Texture](#)
- [How to: Initialize a Texture Programmatically](#)
- [How to: Create a Constant Buffer](#)

Prerequisites

We assume that you are familiar with C++. You also need basic experience with graphics programming concepts.

Instructions

Step 1: Specify dynamic usage

If you want your app to be able to make changes to resources, you must specify those resources as dynamic and writable when you create them.

To specify dynamic usage

1. Specify the resource usage as dynamic. For example, specify the [D3D11_USAGE_DYNAMIC](#) value in the [Usage](#) member of [D3D11_BUFFER_DESC](#) for a vertex or constant buffer and specify the [D3D11_USAGE_DYNAMIC](#) value in the [Usage](#) member of [D3D11_TEXTURE2D_DESC](#) for a 2D texture.
2. Specify the resource as writable. For example, specify the [D3D11_CPU_ACCESS_WRITE](#) value in the [CPUAccessFlags](#) member of [D3D11_BUFFER_DESC](#) or [D3D11_TEXTURE2D_DESC](#).
3. Pass the resource description to the creation function. For example, pass the address of [D3D11_BUFFER_DESC](#) to [ID3D11Device::CreateBuffer](#) and pass the address of [D3D11_TEXTURE2D_DESC](#) to [ID3D11Device::CreateTexture2D](#).

Here is an example of creating a dynamic vertex buffer:

```

// Create a vertex buffer for a triangle.

float2 triangleVertices[] =
{
    float2(-0.5f, -0.5f),
    float2(0.0f, 0.5f),
    float2(0.5f, -0.5f),
};

D3D11_BUFFER_DESC vertexBufferDesc = { 0 };
vertexBufferDesc.ByteWidth = sizeof(float2)* ARRAYSIZE(triangleVertices);
vertexBufferDesc.Usage = D3D11_USAGE_DYNAMIC;
vertexBufferDesc.BindFlags = D3D11_BIND_VERTEX_BUFFER;
vertexBufferDesc.CPUAccessFlags = D3D11_CPU_ACCESS_WRITE;
vertexBufferDesc.MiscFlags = 0;
vertexBufferDesc.StructureByteStride = 0;

D3D11_SUBRESOURCE_DATA vertexBufferData;
vertexBufferData.pSysMem = triangleVertices;
vertexBufferData.SysMemPitch = 0;
vertexBufferData.SysMemSlicePitch = 0;

DX::ThrowIfFailed(
    m_d3dDevice->CreateBuffer(
        &vertexBufferDesc,
        &vertexBufferData,
        &vertexBuffer2
    )
);

```

Step 2: Change a dynamic resource

If you specified a resource as dynamic and writable when you created it, you can later make changes to that resource.

To change data in a dynamic resource

1. Set up the new data for the resource. Declare a [D3D11_MAPPED_SUBRESOURCE](#) type variable and initialize it to zero. You'll use this variable to change the resource.
2. Disable graphics processing unit (GPU) access to the data that you want to change and get a pointer to the memory that contains the data. To get this pointer, pass [D3D11_MAP_WRITE_DISCARD](#) to the *MapType* parameter when you call [ID3D11DeviceContext::Map](#). Set this pointer to the address of the [D3D11_MAPPED_SUBRESOURCE](#) variable from the previous step.
3. Write the new data to the memory.
4. Call [ID3D11DeviceContext::Unmap](#) to reenable GPU access to the data when you're finished writing the new data.

Here is an example of changing a dynamic vertex buffer:

```

// This might get called as the result of a click event to double the size of the triangle.
void TriangleRenderer::MapDoubleVertices()
{
    D3D11_MAPPED_SUBRESOURCE mappedResource;
    ZeroMemory(&mappedResource, sizeof(D3D11_MAPPED_SUBRESOURCE));
    float2 vertices[] =
    {
        float2(-1.0f, -1.0f),
        float2(0.0f, 1.0f),
        float2(1.0f, -1.0f),
    };
    // Disable GPU access to the vertex buffer data.
    auto m_d3dContext = m_deviceResources->GetD3DDeviceContext();
    m_d3dContext->Map(vertexBuffer2.Get(), 0, D3D11_MAP_WRITE_DISCARD, 0, &mappedResource);
    // Update the vertex buffer here.
    memcpy(mappedResource.pData, vertices, sizeof(vertices));
    // Reenable GPU access to the vertex buffer data.
    m_d3dContext->Unmap(vertexBuffer2.Get(), 0);
}

```

Remarks

Using dynamic textures

We recommend that you create only one dynamic texture per format and possibly per size. We don't recommend creating dynamic mipmaps, cubes, and volumes because of the additional overhead in mapping every level. For mipmaps, you can specify [D3D11_MAP_WRITE_DISCARD](#) only on the top level. All levels are discarded by mapping just the top level. This behavior is the same for volumes and cubes. For cubes, the top level and face 0 are mapped.

Using dynamic buffers

When you call [Map](#) on a static vertex buffer while the GPU is using the buffer, you get a significant performance penalty. In this situation, [Map](#) must wait until the GPU is finished reading vertex or index data from the buffer before [Map](#) can return to the calling app, which causes a significant delay. Calling [Map](#) and then rendering from a static buffer several times per frame also prevents the GPU from buffering rendering commands because it must finish commands before returning the map pointer. Without buffered commands, the GPU remains idle until after the app is finished filling the vertex buffer or index buffer and issues a rendering command.

Ideally the vertex or index data would never change, but this is not always possible. There are many situations where the app needs to change vertex or index data every frame, perhaps even multiple times per frame. For these situations, we recommend that you create the vertex or index buffer with [D3D11_USAGE_DYNAMIC](#). This usage flag causes the runtime to optimize for frequent map operations. [D3D11_USAGE_DYNAMIC](#) is only useful when the buffer is mapped frequently; if data is to remain constant, place that data in a static vertex or index buffer.

To receive a performance improvement when you use dynamic vertex buffers, your app must call [Map](#) with the appropriate [D3D11_MAP](#) values. [D3D11_MAP_WRITE_DISCARD](#) indicates that the app doesn't need to keep the old vertex or index data in the buffer. If the GPU is still using the buffer when you call [Map](#) with [D3D11_MAP_WRITE_DISCARD](#), the runtime returns a pointer to a new region of memory instead of the old buffer data. This allows the GPU to continue using the old data while the app places data in the new buffer. No additional memory management is required in the app; the old buffer is reused or destroyed automatically when the GPU is finished with it.

NOTE

When you map a buffer with [D3D11_MAP_WRITE_DISCARD](#), the runtime always discards the entire buffer. You can't preserve info in unmapped areas of the buffer by specifying a nonzero offset or limited size field.

There are cases where the amount of data the app needs to store per map is small, such as adding four vertices to render a sprite. [D3D11_MAP_WRITE_NO_OVERWRITE](#) indicates that the app will not overwrite data already in use in the dynamic buffer. The [Map](#) call will return a pointer to the old data, which will allow the app to add new data in unused regions of the vertex or index buffer. The app must not modify vertices or indices that are used in a draw operation as they might still be in use by the GPU. We recommend that the app then use [D3D11_MAP_WRITE_DISCARD](#) after the dynamic buffer is full to receive a new region of memory, which discards the old vertex or index data after the GPU is finished.

The asynchronous query mechanism is useful to determine if vertices are still in use by the GPU. Issue a query of type [D3D11_QUERY_EVENT](#) after the last draw call that uses the vertices. The vertices are no longer in use when [ID3D11DeviceContext::GetData](#) returns S_OK. When you map a buffer with [D3D11_MAP_WRITE_DISCARD](#) or no map values, you are always guaranteed that the vertices are synchronized properly with the GPU. But, when you map without map values, you will incur the performance penalty described earlier.

NOTE

The Direct3D 11.1 runtime, which is available starting with Windows 8, enables mapping dynamic constant buffers and shader resource views (SRVs) of dynamic buffers with [D3D11_MAP_WRITE_NO_OVERWRITE](#). The Direct3D 11 and earlier runtimes limited no-overwrite partial-update mapping to vertex or index buffers. To determine if a Direct3D device supports these features, call [ID3D11Device::CheckFeatureSupport](#) with [D3D11_FEATURE_D3D11_OPTIONS](#). [CheckFeatureSupport](#) fills members of a [D3D11_FEATURE_DATA_D3D11_OPTIONS](#) structure with the device's features. The relevant members here are [MapNoOverwriteOnDynamicConstantBuffer](#) and [MapNoOverwriteOnDynamicBufferSRV](#).

Related topics

[How to Use Direct3D 11](#)

How To: Create a Compute Shader

11/2/2020 • 3 minutes to read • [Edit Online](#)

A compute shader is an Microsoft High Level Shader Language (HLSL) programmable shader that uses generalized input and output memory access to support virtually any type of calculation. This topic shows how to create a compute shader. The compute shader technology is also known as the DirectCompute technology.

To create a compute shader:

1. Compile the HLSL shader code by calling [D3DCompileFromFile](#).

```
UINT flags = D3DCOMPILE_ENABLE_STRICTNESS;
#if defined( DEBUG ) || defined( _DEBUG )
    flags |= D3DCOMPILE_DEBUG;
#endif
    // Prefer higher CS shader profile when possible as CS 5.0 provides better performance on 11-
    class hardware.
    LPCSTR profile = ( device->GetFeatureLevel() >= D3D_FEATURE_LEVEL_11_0 ) ? "cs_5_0" : "cs_4_0";
    const D3D_SHADER_MACRO defines[] =
    {
        "EXAMPLE_DEFINE", "1",
        NULL, NULL
    };
    ID3DBlob* shaderBlob = nullptr;
    ID3DBlob* errorBlob = nullptr;
    HRESULT hr = D3DCompileFromFile( srcFile, defines, D3D_COMPILE_STANDARD_FILE_INCLUDE,
                                    entryPoint, profile,
                                    flags, 0, &shaderBlob, &errorBlob );
```

2. Create a compute shader using [ID3D11Device::CreateComputeShader](#).

```
ID3D11ComputeShader* g_pFinalPassCS = NULL;
pd3dDevice->CreateComputeShader( pBlobFinalPassCS->GetBufferPointer(),
                                  pBlobFinalPassCS->GetBufferSize(),
                                  NULL, &g_pFinalPassCS );
```

The following code example shows how to compile and create a compute shader.

NOTE

For this example code, you need the Windows SDK 8.0 and the d3dcompiler_44.dll file from the %PROGRAM_FILE%\Windows Kits\8.0\Redist\DXGI\ folder in your path.

```
#define _WIN32_WINNT 0x600
#include <stdio.h>

#include <d3d11.h>
#include <d3dcompiler.h>

#pragma comment(lib,"d3d11.lib")
#pragma comment(lib,"d3dcompiler.lib")

HRESULT CompileComputeShader( _In_ LPCWSTR srcFile, _In_ LPCSTR entryPoint,
```

```

    _In_ ID3D11Device* device, _Outptr_ ID3DBlob** blob )
{
    if ( !srcFile || !entryPoint || !device || !blob )
        return E_INVALIDARG;

    *blob = nullptr;

    UINT flags = D3DCOMPILE_ENABLE_STRICTNESS;
#ifndef defined( DEBUG ) || defined( _DEBUG )
    flags |= D3DCOMPILE_DEBUG;
#endif

    // We generally prefer to use the higher CS shader profile when possible as CS 5.0 is better performance
    // on 11-class hardware
    LPCSTR profile = ( device->GetFeatureLevel() >= D3D_FEATURE_LEVEL_11_0 ) ? "cs_5_0" : "cs_4_0";

    const D3D_SHADER_MACRO defines[] =
    {
        "EXAMPLE_DEFINE", "1",
        NULL, NULL
    };

    ID3DBlob* shaderBlob = nullptr;
    ID3DBlob* errorBlob = nullptr;
    HRESULT hr = D3DCompileFromFile( srcFile, defines, D3D_COMPILE_STANDARD_FILE_INCLUDE,
                                    entryPoint, profile,
                                    flags, 0, &shaderBlob, &errorBlob );
    if ( FAILED(hr) )
    {
        if ( errorBlob )
        {
            OutputDebugStringA( (char*)errorBlob->GetBufferPointer() );
            errorBlob->Release();
        }

        if ( shaderBlob )
            shaderBlob->Release();
    }

    return hr;
}

*blob = shaderBlob;

return hr;
}

int main()
{
    // Create Device
    const D3D_FEATURE_LEVEL lvl[] = { D3D_FEATURE_LEVEL_11_1, D3D_FEATURE_LEVEL_11_0,
                                      D3D_FEATURE_LEVEL_10_1, D3D_FEATURE_LEVEL_10_0 };

    UINT createDeviceFlags = 0;
#ifndef defined( _DEBUG )
    createDeviceFlags |= D3D11_CREATE_DEVICE_DEBUG;
#endif

    ID3D11Device* device = nullptr;
    HRESULT hr = D3D11CreateDevice( nullptr, D3D_DRIVER_TYPE_HARDWARE, nullptr, createDeviceFlags, lvl,
                                   _countof(lvl),
                                   D3D11_SDK_VERSION, &device, nullptr, nullptr );
    if ( hr == E_INVALIDARG )
    {
        // DirectX 11.0 Runtime doesn't recognize D3D_FEATURE_LEVEL_11_1 as a valid value
        hr = D3D11CreateDevice( nullptr, D3D_DRIVER_TYPE_HARDWARE, nullptr, 0, &lvl[1], _countof(lvl) - 1,
                               D3D11_SDK_VERSION, &device, nullptr, nullptr );
    }

    if ( FAILED(hr) )

```

```

    {
        printf("Failed creating Direct3D 11 device %08X\n", hr );
        return -1;
    }

    // Verify compute shader is supported
    if ( device->GetFeatureLevel() < D3D_FEATURE_LEVEL_11_0 )
    {
        D3D11_FEATURE_DATA_D3D10_X_HARDWARE_OPTIONS hwopts = { 0 } ;
        (void)device->CheckFeatureSupport( D3D11_FEATURE_D3D10_X_HARDWARE_OPTIONS, &hwopts, sizeof(hwopts)
    );
        if ( !hwopts.ComputeShaders_Plus_RawAndStructuredBuffers_Via_Shader_4_x )
        {
            device->Release();
            printf( "DirectCompute is not supported by this device\n" );
            return -1;
        }
    }

    // Compile shader
    ID3DBlob *csBlob = nullptr;
    hr = CompileComputeShader( L"ExampleCompute.hlsl", "CSMain", device, &csBlob );
    if ( FAILED(hr) )
    {
        device->Release();
        printf("Failed compiling shader %08X\n", hr );
        return -1;
    }

    // Create shader
    ID3D11ComputeShader* computeShader = nullptr;
    hr = device->CreateComputeShader( csBlob->GetBufferPointer(), csBlob->GetBufferSize(), nullptr,
&computeShader );

    csBlob->Release();

    if ( FAILED(hr) )
    {
        device->Release();
    }

    printf("Success\n");

    // Clean up
    computeShader->Release();

    device->Release();

    return 0;
}

```

The preceding code example compiles the compute shader code in the ExampleCompute.hlsl file. Here is the code in ExampleCompute.hlsl:

```

//-----
// File: BasicCompute11.hlsl
//
// This file contains the Compute Shader to perform array A + array B
//
// Copyright (c) Microsoft Corporation. All rights reserved.
//-----

#ifndef USE_STRUCTURED_BUFFERS

struct BufType
{

```

```

    int i;
    float f;
#ifdef TEST_DOUBLE
    double d;
#endif
};

StructuredBuffer<BufType> Buffer0 : register(t0);
StructuredBuffer<BufType> Buffer1 : register(t1);
RWStructuredBuffer<BufType> BufferOut : register(u0);

[numthreads(1, 1, 1)]
void CSMain( uint3 DTid : SV_DispatchThreadID )
{
    BufferOut[DTid.x].i = Buffer0[DTid.x].i + Buffer1[DTid.x].i;
    BufferOut[DTid.x].f = Buffer0[DTid.x].f + Buffer1[DTid.x].f;
#ifdef TEST_DOUBLE
    BufferOut[DTid.x].d = Buffer0[DTid.x].d + Buffer1[DTid.x].d;
#endif
}

#else // The following code is for raw buffers

ByteAddressBuffer Buffer0 : register(t0);
ByteAddressBuffer Buffer1 : register(t1);
RWByteAddressBuffer BufferOut : register(u0);

[numthreads(1, 1, 1)]
void CSMain( uint3 DTid : SV_DispatchThreadID )
{
#ifdef TEST_DOUBLE
    int i0 = asint( Buffer0.Load( DTid.x*16 ) );
    float f0 = asfloat( Buffer0.Load( DTid.x*16+4 ) );
    double d0 = asdouble( Buffer0.Load( DTid.x*16+8 ), Buffer0.Load( DTid.x*16+12 ) );
    int i1 = asint( Buffer1.Load( DTid.x*16 ) );
    float f1 = asfloat( Buffer1.Load( DTid.x*16+4 ) );
    double d1 = asdouble( Buffer1.Load( DTid.x*16+8 ), Buffer1.Load( DTid.x*16+12 ) );

    BufferOut.Store( DTid.x*16, asuint(i0 + i1) );
    BufferOut.Store( DTid.x*16+4, asuint(f0 + f1) );

    uint dl, dh;
    asuint( d0 + d1, dl, dh );
    BufferOut.Store( DTid.x*16+8, dl );
    BufferOut.Store( DTid.x*16+12, dh );
#else
    int i0 = asint( Buffer0.Load( DTid.x*8 ) );
    float f0 = asfloat( Buffer0.Load( DTid.x*8+4 ) );
    int i1 = asint( Buffer1.Load( DTid.x*8 ) );
    float f1 = asfloat( Buffer1.Load( DTid.x*8+4 ) );

    BufferOut.Store( DTid.x*8, asuint(i0 + i1) );
    BufferOut.Store( DTid.x*8+4, asuint(f0 + f1) );
#endif // TEST_DOUBLE
}

#endif // USE_STRUCTURED_BUFFERS

```

Related topics

- [Compute Shader Overview](#)
- [How to Use Direct3D 11](#)
- [BasicCompute11 sample application](#)

How To: Design a Hull Shader

11/2/2020 • 3 minutes to read • [Edit Online](#)

A hull shader is the first of three stages that work together to implement [tessellation](#) (the other two stages are the tessellator and a domain shader). This topics shows how to design a hull shader.

A hull shader requires two functions, the main hull shader and a patch constant function. The hull shader implements calculations on each control point; the hull shader also calls the patch constant function which implements calculations on each patch.

After you have designed a hull shader, see [How To: Create a Hull Shader](#) to learn how to create a hull shader.

To design a hull shader

1. Define hull shader input control and output control points.

```
// Input control point
struct VS_CONTROL_POINT_OUTPUT
{
    float3 vPosition : WORLDPOS;
    float2 vUV       : TEXCOORD0;
    float3 vTangent  : TANGENT;
};

// Output control point
struct BEZIER_CONTROL_POINT
{
    float3 vPosition   : BEZIERPOS;
};
```

2. Define output patch constant data.

```
// Output patch constant data.
struct HS_CONSTANT_DATA_OUTPUT
{
    float Edges[4]      : SV_TessFactor;
    float Inside[2]     : SV_InsideTessFactor;

    float3 vTangent[4]  : TANGENT;
    float2 vUV[4]        : TEXCOORD;
    float3 vTanUCorner[4] : TANUCORNER;
    float3 vTanVCorner[4] : TANVCORNER;
    float4 vCWts         : TANWEIGHTS;
};
```

For a quad domain, [SV_TessFactor](#) defines 4 edge tessellation factors (to tessellate the edges), since the fixed function tessellator needs to know how much to tessellate. The required outputs are different for the triangle and isoline domains.

The fixed function tessellator doesn't look at any other hull shader outputs, such as other patch constant data or any of the control points. The domain shader -- which is invoked for each point the fixed function tessellator generates -- will see as its input all the hull shader's output control points and all the output patch constant data; the shader evaluates the patch at its location.

3. Define a patch constant function. A patch constant function executes once for each patch to calculate any data that is constant for the entire patch (as opposed to per-control point data, which is computed in the

hull shader).

```
#define MAX_POINTS 32

// Patch Constant Function
HS_CONSTANT_DATA_OUTPUT SubDToBezierConstantsHS(
    InputPatch<VS_CONTROL_POINT_OUTPUT, MAX_POINTS> ip,
    uint PatchID : SV_PrimitiveID )
{
    HS_CONSTANT_DATA_OUTPUT Output;

    // Insert code to compute Output here

    return Output;
}
```

Properties of the patch constant function include:

- One input specifies a variable containing a patch id, and is identified by the **SV_PrimitiveID** system value (see [semantics](#) in shader model 4).
- One input parameter is the input control points, declared in **VS_CONTROL_POINT_OUTPUT** in this example. A patch function can see all the input control points for each patch, there are 32 control points per patch in this example.
- As a minimum, the function must calculate per-patch tessellation factors for the tessellator stage which are identified with [SV_TessFactor](#). A quad domain requires four tessellation factors for the edges and two additional factors (identified by [SV_InsideTessFactor](#)) for tessellating the inside of the patch. The fixed function tessellator doesn't look at any other hull shader outputs (such as the patch constant data or any of the control points).
- The outputs are usually defined by a structure and is identified by **HS_CONSTANT_DATA_OUTPUT** in this example; the structure depends on the domain type and would be different for triangle or isoline domains.

A domain shader on the other hand is invoked for each point the fixed function tessellator generates and needs to see the output control points and the output patch constant data (both from the hull shader) to evaluate a patch at its location.

4. Define a hull shader. A hull shader identifies the properties of a patch including a patch constant function. A hull shader is invoked once for each output control point.

```
[domain("quad")]
[partitioning("integer")]
[outputtopology("triangle_cw")]
[outputcontrolpoints(16)]
[patchconstantfunc("SubDToBezierConstantsHS")]
BEZIER_CONTROL_POINT SubDToBezierHS
{
    InputPatch<VS_CONTROL_POINT_OUTPUT, MAX_POINTS> ip,
    uint i : SV_OutputControlPointID,
    uint PatchID : SV_PrimitiveID )
{
    VS_CONTROL_POINT_OUTPUT Output;

    // Insert code to compute Output here.

    return Output;
}
```

A hull shader uses the following attributes:

- A [domain](#) attribute.

- A [partitioning](#) attribute.
- An [outputtopology](#) attribute.
- An [outputcontrolpoints](#) attribute.
- A [patchconstantfunc](#) attribute. A hull shader calculates output control points, there are 16 output Bezier control points in this example.

All the input control points (identified by `VS_CONTROL_POINT_OUTPUT`) are visible to each hull shader invocation. In this example, there are 32 input control points.

A hull shader is invoked once per output control point (identified with `SV_OutputControlPointID`) for each patch (identified with `SV_PrimitiveID`). The purpose of this particular shader is to calculate output i , which was defined to be a BEZIER control point (this example has 16 output control points defined by `outputcontrolpoints`).

A hull shader runs a routine once per patch (the patch constant function) to compute patch-constant data (tessellation factors as a minimum). Separately, a hull shader runs a patch constant function (called `SubDToBezierConstantsHS`) on each patch to compute patch-constant data such as tessellation factors for the tessellator stage.

Related topics

[How to Use Direct3D 11](#)

[Tessellation Overview](#)

How To: Create a Hull Shader

2/22/2020 • 2 minutes to read • [Edit Online](#)

A hull shader is the first of three stages that work together to implement [tessellation](#). The hull-shader outputs drive the tessellator stage, as well as the domain-shader stage. This topic shows how to create a hull shader.

A hull shader transforms a set of input control points (from a vertex shader) into a set of output control points. The number of input and output points can vary in contents and number depending on the transform (a typical transformation would be a basis transformation).

A hull shader also outputs patch constant information, such as tessellation factors, for a domain shader and the tessellator. The fixed-function tessellator stage uses the tessellation factors as well as other state declared in a hull shader to determine how much to tessellate.

To create a hull shader

1. Design a hull shader. See [How To: Design a Hull Shader](#).
2. Compile the shader code
3. Create a hull-shader object using [ID3D11Device::CreateHullShader](#).

```
HRESULT CreateHullShader(  
    const void *pShaderBytecode,  
    SIZE_T BytecodeLength,  
    ID3D11ClassLinkage *pClassLinkage,  
    ID3D11HullShader **ppHullShader  
) ;
```

4. Initialize the pipeline stage using [ID3D11DeviceContext::HSSetShader](#).

```
void HSSetShader(  
    ID3D11HullShader *pHullShader,  
    ID3D11ClassInstance *const *ppClassInstances,  
    UINT NumClassInstances  
) ;
```

A domain shader must be bound to the pipeline if a hull shader is bound. In particular, it is not valid to directly stream out hull shader control points with the geometry shader.

Related topics

[How to Use Direct3D 11](#)

[Tessellation Overview](#)

How To: Initialize the Tessellator Stage

2/22/2020 • 2 minutes to read • [Edit Online](#)

In general, tessellation expands the compact, user-defined, model of a patch into geometry that contains a programmable amount of detail. The geometry is typically a set of triangles that represents detailed surface geometry. This topic shows how to initialize the tessellator stage.

The tessellator stage is the second of three stages that work together to tessellate or tile a surface. The first stage is the hull-shader stage; it operates once per patch and configures how the next stage (the fixed function tessellator) behaves. A hull shader also generates user-defined outputs such as output-control points and patch constants that are sent past the tessellator directly to the third stage, the domain-shader stage. A domain shader is invoked once per tessellator-stage point and evaluates surface positions.

The tessellator stage is a fixed function stage, there is no shader to generate, and no state to set. It receives all of its setup state from the hull-shader stage; once the hull-shader stage has been initialized, the tessellator stage is automatically initialized.

To initialize the tessellator stage

- Initialize the hull-shader stage using [ID3D11DeviceContext::HSSetShader](#).

```
void HSSetShader(  
    ID3D11HullShader *pHullShader,  
    ID3D11ClassInstance *const *ppClassInstances,  
    UINT NumClassInstances  
) ;
```

ppClassInstances is a pointer to an array of shader interfaces, represented by [ID3D11ClassInstance](#) pointers and the number of interfaces, represented by *NumClassInstances*. If not used, these parameters can be set to **NULL** and 0 respectively.

After the hull-shader stage is initialized, you should also initialize the domain-shader stage.

Related topics

[How to Use Direct3D 11](#)

[Tessellation Overview](#)

How To: Design a Domain Shader

11/2/2020 • 2 minutes to read • [Edit Online](#)

A domain shader is the third of three stages that work together to implement [tessellation](#). The domain shader generates the surface geometry from the transformed control points from a hull shader and the UV coordinates. This topics shows how to design a domain shader.

A domain shader is invoked once for each point generated by the fixed function tessellator. The inputs are the UV[W] coordinates of the point on the patch, as well as all of the output data from the hull shader including control points and patch constants. The output is a vertex defined in whatever way is desired. If the output is being sent to the pixel shader, the output must include a position (denoted with a SV_Position semantic).

To design a domain shader

1. Define the domain attribute.

```
[domain("quad")]
```

The domain is defined for a quad patch.

2. Declare the location on the hull with the [domain location](#) system value.

- For a quad patch, use a float2.
- For a tri patch, use a float3 (for barycentric coordinates)
- For an isoline, use a float2.

Therefore, the domain location for a quad patch looks like this:

```
float2 UV : SV_DomainLocation
```

3. Define the other inputs.

The other inputs come from the hull shader and are user defined. This includes the input control points for patch, of which there can be between 1 and 32 points, and input patch constant data.

The control points are user defined, usually with a structure such as this one (defined in [How To: Design a Hull Shader](#)):

```
const OutputPatch<BEZIER_CONTROL_POINT, 16> bezpatch
```

The patch constant data is also user defined, and might look like this one (defined in [How To: Design a Hull Shader](#)):

```
HS_CONSTANT_DATA_OUTPUT input
```

4. Add user-defined code to compute the outputs; this makes up the body of the domain shader.

This structure contains user-defined domain shader outputs.

```

struct DS_OUTPUT
{
    float3 vNormal : NORMAL;
    float2 vUV : TEXCOORD;
    float3 vTangent : TANGENT;
    float3 vBiTangent : BITANGENT;

    float4 vPosition : SV_POSITION;
};

```

The function takes each input UV (from the tessellator) and evaluates the Bezier patch at this position.

```

[domain("quad")]
DS_OUTPUT BezierEvalDS( HS_CONSTANT_DATA_OUTPUT input,
                        float2 UV : SV_DomainLocation,
                        const OutputPatch<BEZIER_CONTROL_POINT, 16> bezpatch )
{
    DS_OUTPUT Output;

    // Insert code to compute the output here.

    return Output;
}

```

The function is invoked once for each point generated by the fixed function tessellator. Since this example uses a quad patch, the input domain location ([SV_DomainLocation](#)) is a float2 (UV); a tri patch would have a float3 input location (UVW barycentric coordinates), and an isoline would have a float2 input domain location.

The other inputs for the function come from the hull shader directly. In this example it is 16 control points each one being a **BEZIER_CONTROL_POINT**, as well as patch constant data ([HS_CONSTANT_DATA_OUTPUT](#)). The output is a vertex containing any data desired - **DS_OUTPUT** in this example.

After designing a domain shader, see [How To: Create a Domain Shader](#).

Related topics

[How to Use Direct3D 11](#)

[Tessellation Overview](#)

How To: Create a Domain Shader

2/22/2020 • 2 minutes to read • [Edit Online](#)

A domain shader is the third of three stages that work together to implement [tessellation](#). The inputs for the domain-shader stage come from a hull shader. This topic shows how to create a domain shader.

A domain shader transforms surface geometry (created by the fixed-function tessellator stage) using hull shader output-control points, hull shader output patch-constant data, and a single set of tessellator uv coordinates.

To create a domain shader

1. Design a domain shader. See [How To: Design a Domain Shader](#).
2. Compile the shader code.
3. Create a domain-shader object using [ID3D11Device::CreateDomainShader](#).

```
HRESULT CreateDomainShader(
    const void *pShaderBytecode, //
    SIZE_T BytecodeLength, //
    ID3D11ClassLinkage *pClassLinkage, //
    ID3D11DomainShader **ppDomainShader
);
```

4. Initialize the pipeline stage using [ID3D11DeviceContext::DSSetShader](#).

```
void DSSetShader(
    ID3D11DomainShader *pDomainShader, //
    ID3D11ClassInstance *const *ppClassInstances,
    UINT NumClassInstances
);
```

A domain shader must be bound to the pipeline if a hull shader is bound. In particular, it is not valid to directly stream out hull shader control points with the geometry shader.

Related topics

[How to Use Direct3D 11](#)

[Tessellation Overview](#)

How To: Compile a Shader

11/2/2020 • 3 minutes to read • [Edit Online](#)

You typically use the [fxc.exe](#) HLSL code compiler as part of the build process to compile shader code. For more info about this, see [Compiling Shaders](#). This topic shows how to use the [D3DCompileFromFile](#) function at run time to compile shader code.

To compile a shader:

- Compile HLSL shader code by calling [D3DCompileFromFile](#).

```
UINT flags = D3DCOMPILE_ENABLE_STRICTNESS;
#if defined( DEBUG ) || defined( _DEBUG )
    flags |= D3DCOMPILE_DEBUG;
#endif
    // Prefer higher CS shader profile when possible as CS 5.0 provides better performance on 11-
    class hardware.
    LPCSTR profile = ( device->GetFeatureLevel() >= D3D_FEATURE_LEVEL_11_0 ) ? "cs_5_0" : "cs_4_0";
const D3D_SHADER_MACRO defines[] =
{
    "EXAMPLE_DEFINE", "1",
    NULL, NULL
};
ID3DBlob* shaderBlob = nullptr;
ID3DBlob* errorBlob = nullptr;
HRESULT hr = D3DCompileFromFile( srcFile, defines, D3D_COMPILE_STANDARD_FILE_INCLUDE,
                                entryPoint, profile,
                                flags, 0, &shaderBlob, &errorBlob );
```

The following code example shows how to compile various shaders.

NOTE

For this example code, you need the Windows SDK 8.0 and the d3dcompiler_44.dll file from the %PROGRAM_FILE%\Windows Kits\8.0\Redist\DXGI\ folder in your path. Windows Store apps support run time compilation for development but not for deployment.

```
#define _WIN32_WINNT 0x600
#include <stdio.h>

#include <d3dcompiler.h>

#pragma comment(lib,"d3dcompiler.lib")

HRESULT CompileShader( _In_ LPCWSTR srcFile, _In_ LPCSTR entryPoint, _In_ LPCSTR profile, _Outptr_
ID3DBlob** blob )
{
    if ( !srcFile || !entryPoint || !profile || !blob )
        return E_INVALIDARG;

    *blob = nullptr;

    UINT flags = D3DCOMPILE_ENABLE_STRICTNESS;
#if defined( DEBUG ) || defined( _DEBUG )
    flags |= D3DCOMPILE_DEBUG;
```

```

#endif

const D3D_SHADER_MACRO defines[] =
{
    "EXAMPLE_DEFINE", "1",
    NULL, NULL
};

ID3DBlob* shaderBlob = nullptr;
ID3DBlob* errorBlob = nullptr;
HRESULT hr = D3DCompileFromFile( srcFile, defines, D3D_COMPILE_STANDARD_FILE_INCLUDE,
                                entryPoint, profile,
                                flags, 0, &shaderBlob, &errorBlob );
if ( FAILED(hr) )
{
    if ( errorBlob )
    {
        OutputDebugStringA( (char*)errorBlob->GetBufferPointer() );
        errorBlob->Release();
    }

    if ( shaderBlob )
        shaderBlob->Release();

    return hr;
}

*blob = shaderBlob;

return hr;
}

int main()
{
    // Compile vertex shader shader
    ID3DBlob *vsBlob = nullptr;
    HRESULT hr = CompileShader( L"BasicHLSL11_VS.hlsl", "VSMain", "vs_4_0_level_9_1", &vsBlob );
    if ( FAILED(hr) )
    {
        printf("Failed compiling vertex shader %08X\n", hr );
        return -1;
    }

    // Compile pixel shader shader
    ID3DBlob *psBlob = nullptr;
    hr = CompileShader( L"BasicHLSL11_PS.hlsl", "PSMain", "ps_4_0_level_9_1", &psBlob );
    if ( FAILED(hr) )
    {
        vsBlob->Release();
        printf("Failed compiling pixel shader %08X\n", hr );
        return -1;
    }

    printf("Success\n");

    // Clean up
    vsBlob->Release();
    psBlob->Release();

    return 0;
}

```

The preceding code example compiles the pixel and vertex shader code blocks in the BasicHLSL11_PS.hlsl and BasicHLSL11_VS.hlsl files. Here is the code in BasicHLSL11_PS.hlsl:

```

//-----
// File: BasicHLSL11_PS.hlsl
//
// The pixel shader file for the BasicHLSL11 sample.
//
// Copyright (c) Microsoft Corporation. All rights reserved.
//-----

//-----
// Globals
//-----
cbuffer cbPerObject : register( b0 )
{
    float4      g_vObjectColor       : packoffset( c0 );
};

cbuffer cbPerFrame : register( b1 )
{
    float3      g_vLightDir         : packoffset( c0 );
    float       g_fAmbient          : packoffset( c0.w );
};

//-----
// Textures and Samplers
//-----
Texture2D    g_txDiffuse : register( t0 );
SamplerState g_samLinear : register( s0 );

//-----
// Input / Output structures
//-----
struct PS_INPUT
{
    float3 vNormal      : NORMAL;
    float2 vTexCoord     : TEXCOORD0;
};

//-----
// Pixel Shader
//-----
float4 PSMain( PS_INPUT Input ) : SV_TARGET
{
    float4 vDiffuse = g_txDiffuse.Sample( g_samLinear, Input.vTexCoord );

    float fLighting = saturate( dot( g_vLightDir, Input.vNormal ) );
    fLighting = max( fLighting, g_fAmbient );

    return vDiffuse * fLighting;
}

```

Here is the code in BasicHLSL11_VS.hlsl:

```

//-----
// File: BasicHLSL11_VS.hlsl
//
// The vertex shader file for the BasicHLSL11 sample.
//
// Copyright (c) Microsoft Corporation. All rights reserved.
//-----

//-----
// Globals
//-----
cbuffer cbPerObject : register( b0 )
{
    matrix      g_mWorldViewProjection   : packoffset( c0 );
    matrix      g_mWorld                 : packoffset( c4 );
};

//-----
// Input / Output structures
//-----
struct VS_INPUT
{
    float4 vPosition     : POSITION;
    float3 vNormal       : NORMAL;
    float2 vTexcoord     : TEXCOORD0;
};

struct VS_OUTPUT
{
    float3 vNormal       : NORMAL;
    float2 vTexcoord     : TEXCOORD0;
    float4 vPosition     : SV_POSITION;
};

//-----
// Vertex Shader
//-----
VS_OUTPUT VSMain( VS_INPUT Input )
{
    VS_OUTPUT Output;

    Output.vPosition = mul( Input.vPosition, g_mWorldViewProjection );
    Output.vNormal = mul( Input.vNormal, (float3x3)g_mWorld );
    Output.vTexcoord = Input.vTexcoord;

    return Output;
}

```

Related topics

[How to Use Direct3D 11](#)

How to: Record a Command List

2/22/2020 • 2 minutes to read • [Edit Online](#)

A [command list](#) is a recorded list of rendering commands. This topic shows how to create and record a [command list](#). Use a command list to record render commands and play them back later. A command list is convenient for splitting rendering tasks between threads.

To record a command list

1. A command list must be created from a deferred context, which contains device state and rendering actions. Given a device, create a deferred context by calling [ID3D11Device::CreateDeferredContext](#).

```
HRESULT hr;
ID3D11DeviceContext* pDeferredContext = NULL;

hr = g_pd3dDevice->CreateDeferredContext(0, &pDeferredContext);
```

2. Use the deferred context to render.

```
float ClearColor[4] = { 0.0f, 0.125f, 0.3f, 1.0f };
pDeferredContext->ClearRenderTargetView( g_pRenderTargetView, ClearColor );

// Add additional rendering commands
...
```

This simple example clears a render target, but you could add additional render commands.

3. Create/record a command list by calling [ID3D11DeviceContext::FinishCommandList](#) and passing a pointer to an uninitialized [ID3D11CommandList](#) interface.

```
ID3D11CommandList* pd3dCommandList = NULL;
HRESULT hr;
hr = pDeferredContext->FinishCommandList(FALSE, &pd3dCommandList);
```

When the method returns, a command list is created containing all the render commands and an interface is returned to the application.

The boolean parameter tells the runtime what to do with the pipeline state in the deferred context. **TRUE** means restore the state of the device context to its pre-command list condition when recording is completed, **FALSE** means do not change the state after recording. This means that the device context will reflect the state changes contained in the command list.

To see an example for playing back a command list, see [How to: Play Back a Command List](#).

Related topics

[Command List](#)

[How to Use Direct3D 11](#)

How to: Play Back a Command List

2/22/2020 • 2 minutes to read • [Edit Online](#)

A [command list](#) is a recorded list of rendering commands. Use a command list to pre-record drawing commands and play them back later. This topic shows how to play back a [command list](#). A command list can be used to split rendering tasks between threads.

This section describes how to play back a command list. For recording a command list, see [How to: Record a Command List](#).

To play back a command list

- Call [ID3D11DeviceContext::ExecuteCommandList](#) and pass a valid [ID3D11CommandList](#) object.

```
if(g_pd3dCommandList)
{
    g_pImmediateContext->ExecuteCommandList(g_pd3dCommandList, TRUE);
}
```

[ExecuteCommandList](#) must be executed on the [immediate context](#) for recorded commands to be run on the graphics processing unit (GPU). Use the immediate context to feed commands to the GPU for execution, use a deferred context to record commands for playback onto another command list. When you call [ExecuteCommandList](#) onto another deferred context, you create a 'merged' deferred command list. To run the commands on the merged deferred command list, you need to execute them on the immediate context.

Related topics

[Command List](#)

[How to Use Direct3D 11](#)

How To: Check for Driver Support

2/22/2020 • 2 minutes to read • [Edit Online](#)

This topic shows how to determine whether multithreading features (including [resource creation](#) and [command lists](#)) are supported for hardware acceleration.

We recommend for applications to check for graphics hardware support of multithreading. If the driver and graphics hardware do not support multithreaded object creation, performance can be limited in the following ways:

- Creating multiple objects (even of different types) at the same time might be limited.
- Creating an object while rendering graphics commands by using an [immediate context](#) might be limited. For example, if hardware does not support multithreading, an application should avoid creating on a background thread an object that requires a very long time to create. A create operation that takes very long can block immediate context rendering and increase the risk of a visual frame rate stutter.

The runtime supports multithreading and command lists regardless of driver and hardware support; if there is no driver and hardware support for either multithreads or command lists, the runtime will emulate the functionality. For more information about multithreading, see [Introduction to Multithreading in Direct3D 11](#).

To check for driver support for multithreading:

1. Initialize an [ID3D11Device](#) interface object. By default, multithreading is enabled.
2. Call [ID3D11Device::CheckFeatureSupport](#). Pass the `D3D11_FEATURE_THREADING` value to the `Feature` parameter, pass the [D3D11_FEATURE_DATA_THREADING](#) structure to the `pFeatureSupportData` parameter, and pass the size of the [D3D11_FEATURE_DATA_THREADING](#) structure to the `FeatureSupportDataSize` parameter.
3. If the [ID3D11Device::CheckFeatureSupport](#) method succeeds, the [D3D11_FEATURE_DATA_THREADING](#) structure that you passed in the previous step will be initialized with information about multithreading support.
 - If `DriverConcurrentCreates` is `TRUE`, a driver can create more than one resource at the same time (concurrently) on different threads.
If `DriverCommandLists` is `TRUE`, the driver supports command lists. That is, rendering commands issued by an immediate context can be concurrent with object creation on separate threads with low risk of a frame rate stutter.
 - If `DriverConcurrentCreates` is `FALSE`, a driver does not support concurrent creation, which means the amount of concurrency possible is extremely limited. The graphics hardware cannot create objects of different types on different threads simultaneously. Additionally, the graphics hardware cannot use an immediate context to issue render commands while the graphics hardware attempts to create a resource on another thread.

Related topics

[How to Use Direct3D 11](#)

[Multithreading](#)

What's new in Direct3D 11

11/2/2020 • 2 minutes to read • [Edit Online](#)

This section describes features added in Direct3D 11, Direct3D 11.1, and Direct3D 11.2.

Microsoft Direct3D 11 is an extension of the Microsoft Direct3D 10/Microsoft Direct3D 10.1 rendering API. For more introductory material about using Direct3D 11, see the [Programming Guide for Direct3D 10](#).

In this section

TOPIC	DESCRIPTION
Direct3D 11 Features	The programming guide contains information about how to use the Direct3D 11 programmable pipeline to create realtime 3D graphics for games, and for scientific and desktop applications.
Direct3D 11.1 Features	The following functionality has been added in Direct3D 11.1, which is included with Windows 8, Windows RT, and Windows Server 2012. Partial support for Direct3D 11.1 is available on Windows 7 and Windows Server 2008 R2 via the Platform Update for Windows 7 , which is available through the Platform Update for Windows 7 .
Direct3D 11.2 Features	The following functionality has been added in Direct3D 11.2, which is included with Windows 8.1, Windows RT 8.1, and Windows Server 2012 R2.
Direct3D 11.3 Features	The following sections describe the functionality has been added in Direct3D 11.3. These features are also available in Direct3D 12.
Direct3D 11.4 Features	The following functionality has been added in Direct3D 11.4.
Features Introduced In Previous Releases	Discover what new features have been added to the previous SDK updates:

Related topics

[Direct3D 11 Graphics](#)

Direct3D 11 Features

11/2/2020 • 7 minutes to read • [Edit Online](#)

The programming guide contains information about how to use the Direct3D 11 programmable pipeline to create realtime 3D graphics for games, and for scientific and desktop applications.

- [Compute Shader](#)
- [Dynamic Shader Linking](#)
- [Multithreading](#)
- [Tessellation](#)
- [Full Listing of Features](#)
- [Features Added in Previous Releases](#)
- [Related topics](#)

Compute Shader

A compute shader is a programmable shader designed for general-purpose data-parallel processing. In other words, compute shaders allow a GPU to be used as a general-purpose parallel processor. The compute shader is similar to the other programmable pipeline shaders (such as vertex, pixel, geometry) in the way that it accesses inputs and outputs. The compute shader technology is also known as the DirectCompute technology. A compute shader is integrated into Direct3D and is accessible through a Direct3D device. It can directly share memory resources with graphics shaders by using the Direct3D device. However, it is not directly connected to other shader stages.

A compute shader is designed for mass-market applications that perform computations at interactive rates, when the cost of transitioning between the API (and its associated software stack) and a CPU would consume too much overhead.

A compute shader has its own set of states. A compute shader does not necessarily have a forced 1-1 mapping to either input records (like a vertex shader does) or output records (like the pixel shader does). Some features of the graphics shader are supported, but others have been removed so that new compute shader-specific features could be added.

To support the compute shader-specific features, several new resource types are now available, such as read/write buffers, textures, and structured buffers.

See [Compute Shader Overview](#) for additional information.

Dynamic Shader Linking

Rendering systems must deal with significant complexity when they manage shaders, while providing the opportunity to optimize shader code. This becomes an even greater challenge because shaders must support a variety of different materials in a rendered scene across various hardware configurations. To address this challenge, shader developers have often resorted to one of two general approaches. They have either created fully featured large, general-purpose shaders that can be used by a wide variety of scene items, which trade off some performance for flexibility, or created individual shaders for each geometry stream, material type, or light type combination needed.

These large, general-purpose shaders handle this challenge by recompiling the same shader with different preprocessor definitions, and the latter method uses brute-force developer power to achieve the same result. The shader permutation explosion has often been a problem for developers who must now manage thousands

of different shader permutations within their game and asset pipeline.

Direct3D 11 and shader model 5 introduce object-oriented language constructs and provide runtime support of shader linking to help developers program shaders.

See [Dynamic Linking](#) for additional information.

Multithreading

Many graphics applications are CPU-bound because of costly activities such as scene graph traversal, object sorting, and physics simulations. Because multicore systems are becoming increasingly available, Direct3D 11 has improved its multithreading support to enable efficient interaction between multiple CPU threads and the D3D11 graphics APIs.

Direct3D 11 enables the following functionality to support multithreading:

- Concurrent objects are now created in separate threads — Making entry-point functions that create objects free-threaded makes it possible for many threads to create objects simultaneously. For example, an application can now compile a shader or load a texture on one thread while rendering on another.
- Command lists can be created on multiple threads — A command list is a recorded sequence of graphics commands. With Direct3D 11, you can create command lists on multiple CPU threads, which enables parallel traversal of the scene database or physics processing on multiple threads. This frees the main rendering thread to dispatch command buffers to the hardware.

See [MultiThreading](#) for additional information.

Tessellation

Tessellation can be used to render a single model with varying levels of detail. This approach generates a more geometrically accurate model that depends on the level of detail required for a scene. Use tessellation in a scene where the level of detail allows a lower geometry model, which reduces the demand on memory bandwidth consumed during rendering.

In Direct3D, tessellation is implemented on the GPU to calculate a smoother curved surface from a coarse (less detailed) input patch. Each (quad or triangle) patch face is subdivided into smaller triangular faces that better approximate the surface that you want.

For information about implementing tessellation in the graphics pipeline, see [Tessellation Overview](#).

Full Listing of Features

This is a complete list of the features in Direct3D 11.

- You can run Direct3D 11 on [downlevel hardware](#) by specifying a [feature level](#) when you create a device.
- You can perform tessellation (see [Tessellation Overview](#)) by using the following shader types:
 - Hull Shader
 - Domain Shader
- Direct3D 11 supports multithreading (see [MultiThreading](#))
 - Multithread resource/shader/object creation
 - Multithreaded Display list creation
- Direct3D 11 expands shaders with the following features (see [Shader Model 5](#))
 - Addressable resources - textures, constant buffers, and samplers
 - Additional resource types, such as read/write buffers and textures (see [New Resource Types](#)).

- Subroutines
- Compute shader (see [Compute Shader Overview](#)) - A shader that speeds up computations by dividing the problem space between several software threads or groups of threads, and sharing data among shader registers to significantly reduce the amount of data required to input into a shader. Algorithms that the compute shader can significantly improve include post processing, animation, physics, and artificial intelligence.
- Geometry shader (see [Geometry Shader Features](#))
 - Instancing - Allows the geometry shader to output a maximum of 1024 vertices, or any combination of instances and vertices up to 1024 (maximum of 32 instances of 32 vertices each).
- Pixel shader
 - Coverage as PS Input
 - Programmable Interpolation of Inputs - The pixel shader can evaluate attributes within the pixel, anywhere on the multisample grid
 - Centroid sampling of attributes must obey the following rules:
 - If all samples in the primitive are covered, the attribute is evaluated at the pixel center regardless of whether the sample pattern has a sample location at the pixel center.
 - Otherwise, the attribute is evaluated at the first covered sample, that is, the sample with the lowest index among all sample indexes. In this situation, sample coverage is determined after applying the logical AND operation to the coverage and the sample-mask rasterizer state.
 - If no samples are covered (such as on helper pixels that are executed off the bounds of a primitive to fill out 2x2 pixel stamps), the attribute is evaluated in one of the following ways:
 - If the sample-mask rasterizer state is a subset of the samples in the pixel, the first sample that is covered by the sample-mask rasterizer state is the evaluation point.
 - Otherwise, in the full sample-mask condition, the pixel center is the evaluation point.
- Direct3D 11 expands textures (see [Textures Overview](#)) with the following features
 - Gather4
 - Support for multi-component textures - specify a channel to load from
 - Support for programmable offsets
 - Streaming
 - Texture clamps to limit WDDM preload
 - 16K texture limits
 - Require 8-bits of subtexel and sub-mip precision on texture filtering
 - New texture compression formats (1 new LDR format and 1 new HDR format)
- Direct3D 11 supports conservative oDepth - This algorithm allows a pixel shader to compare the per-pixel depth value of the pixel shader with that in the rasterizer. The result enables early depth culling operations while maintaining the ability to output oDepth from a pixel shader.

- Direct3D 11 supports large memory
 - Allow for resources > 4GB
 - Keep indices of resources 32bit, but resource larger
- Direct3D 11 supports stream output improvements
 - Addressable Stream output
 - Increase Stream output count to 4
 - Change all stream output buffers to be multi-element
- Direct3D 11 supports Shader Model 5 (see [Shader Model 5](#))
 - Doubles with denorms
 - Count bits set instruction
 - Find first bit set instruction
 - Carry/Overflow handling
 - Bit reversal instructions for FFTs
 - Conditional Swap intrinsic
 - Resinfo on buffers
 - Reduced-precision reciprocal
 - Shader conversion instructions - fp16 to fp32 and vice versa
 - Structured buffer, which is a new type of buffer containing structured elements.
- Direct3D 11 supports read-only depth or stencil views
 - Disables writes to the part that is read-only, allows for using texture as input and for depth-culling
- Direct3D 11 supports draw indirect - Direct3D 10 implements DrawAuto, which takes content (generated by the GPU) and renders it (on the GPU). Direct3D 11 generalizes DrawAuto so that it can be called by a Compute Shader using DrawInstanced and DrawIndexedInstanced.
- Direct3D 11 supports miscellaneous features
 - Floating-point viewports
 - Per-resource mipmap clamping
 - Depth Bias - This algorithm updates the behavior of depth bias, by using rasterizer state. The result eliminates the scenarios where the calculated bias could be NaN.
 - Resource limits - Resource indices are still required to be ≤ 32 bits, but resources can be larger than 4 GB.
 - Rasterizer Precision
 - MSAA Requirements
 - Counters Reduced
 - 1-Bit Format and Text Filter Removed

Features Added in Previous Releases

For the list of the features added in previous releases, see the following topics:

- [What's New in the August 2009 Windows 7/Direct3D 11 SDK](#)
- [What's New in the November 2008 Direct3D 11 Technical Preview](#)

Related topics

[What's new in Direct3D 11](#)

Direct3D 11.1 Features

11/2/2020 • 12 minutes to read • [Edit Online](#)

The following functionality has been added in Direct3D 11.1, which is included with Windows 8, Windows RT, and Windows Server 2012. Partial support for [Direct3D 11.1](#) is available on Windows 7 and Windows Server 2008 R2 via the [Platform Update for Windows 7](#), which is available through the [Platform Update for Windows 7](#).

- [Shader tracing and compiler enhancements](#)
- [Direct3D device sharing](#)
- [Check support of new Direct3D 11.1 features and formats](#)
- [Use HLSL minimum precision](#)
- [Specify user clip planes in HLSL on feature level 9 and higher](#)
- [Create larger constant buffers than a shader can access](#)
- [Use logical operations in a render target](#)
- [Force the sample count to create a rasterizer state](#)
- [Process video resources with shaders](#)
- [Extended support for shared Texture2D resources](#)
- [Change subresources with new copy options](#)
- [Discard resources and resource views](#)
- [Support a larger number of UAVs](#)
- [Bind a subrange of a constant buffer to a shader](#)
- [Retrieve the subrange of a constant buffer that is bound to a shader](#)
- [Clear all or part of a resource view](#)
- [Map SRVs of dynamic buffers with NO_OVERWRITE](#)
- [Use UAVs at every pipeline stage](#)
- [Extended support for WARP devices](#)
- [Use Direct3D in Session 0 processes](#)
- [Support for shadow buffer on feature level 9](#)
- [Related topics](#)

Shader tracing and compiler enhancements

Direct3D 11.1 lets you use shader tracing to ensure that your code is performing as intended and if it isn't you can discover and remedy the problem. The Windows Software Development Kit (SDK) for Windows 8 contains HLSL compiler enhancements. Shader tracing and the HLSL compiler are implemented in D3dcompiler_nn.dll.

The shader tracing API and the enhancements to the HLSL compiler consists of the following methods and functions.

- [ID3D11RefDefaultTrackingOptions::SetTrackingOptions](#)
- [ID3D11RefTrackingOptions::SetTrackingOptions](#)
- [ID3D11TracingDevice::SetShaderTrackingOptions](#)
- [ID3D11TracingDevice::SetShaderTrackingOptionsByType](#)
- [ID3D11ShaderTraceFactory::CreateShaderTrace](#)
- [ID3D11ShaderTrace::TraceReady](#)
- [ID3D11ShaderTrace::ResetTrace](#)

- [ID3D11ShaderTrace::GetTraceStats](#)
- [ID3D11ShaderTrace::PSSelectStamp](#)
- [ID3D11ShaderTrace::GetInitialRegisterContents](#)
- [ID3D11ShaderTrace::GetStep](#)
- [ID3D11ShaderTrace::GetWrittenRegister](#)
- [ID3D11ShaderTrace::GetReadRegister](#)
- [D3DCompile2](#)
- [D3DCompileFromFile](#)
- [D3DDisassemble11Trace](#)
- [D3DDisassembleRegion](#)
- [D3DGetTraceInstructionOffsets](#)
- [D3DReadFileToBlob](#)
- [D3DSetBlobPart](#)
- [D3DWriteBlobToFile](#)

The D3dcompiler.lib library requires D3dcompiler_nn.dll. This DLL is not part of Windows 8; it is in the \bin folder of the Windows SDK for Windows 8 along with the Fxc.exe command-line version of the HLSL compiler.

NOTE

While you can use this library and DLL combination for development, you can't deploy Windows Store apps that use this combination. Therefore, you must instead compile HLSL shaders before you ship your Windows Store app. You can write HLSL compilation binaries to disk, or the compiler can generate headers with static byte arrays that contain the shader blob data. You use the [ID3DBlob](#) interface to access the blob data. To develop your Windows Store app, call [D3DCompile2](#) or [D3DCompileFromFile](#) to compile the raw HLSL source, and then feed the resulting blob data to Direct3D.

Direct3D device sharing

Direct3D 11.1 enables Direct3D 10 APIs and Direct3D 11 APIs to use one underlying rendering device.

This Direct3D 11.1 feature consists of the following methods and interface.

- [ID3D11Device1::CreateDeviceContextState](#)
- [ID3D11DeviceContext1::SwapDeviceContextState](#)
- [ID3DDeviceContextState](#)

Check support of new Direct3D 11.1 features and formats

Direct3D 11.1 lets you check for new features that the graphics driver might support and new ways that a format is supported on a device. Microsoft DirectX Graphics Infrastructure (DXGI) 1.2 also specifies new [DXGI_FORMAT](#) values.

This Direct3D 11.1 feature consists of the following API.

- [ID3D11Device::CheckFeatureSupport](#) with [D3D11_FEATURE_DATA_D3D11_OPTIONS](#), [D3D11_FEATURE_DATA_ARCHITECTURE_INFO](#), [D3D11_FEATURE_DATA_D3D9_OPTIONS](#), [D3D11_FEATURE_DATA_SHADER_MIN_PRECISION_SUPPORT](#), and [D3D11_FEATURE_DATA_D3D9_SHADOW_SUPPORT](#) structures
- [ID3D11Device::CheckFormatSupport](#) with [D3D11_FORMAT_SUPPORT_DECODER_OUTPUT](#), [D3D11_FORMAT_SUPPORT_VIDEO_PROCESSOR_OUTPUT](#),

[D3D11_FORMAT_SUPPORT_VIDEO_PROCESSOR_INPUT](#),
[D3D11_FORMAT_SUPPORT_VIDEO_ENCODER](#), and
[D3D11_FORMAT_SUPPORT2_OUTPUT_MERGER_LOGIC_OP](#)

Use HLSL minimum precision

Starting with Windows 8, graphics drivers can implement minimum precision [HLSL scalar data types](#) by using any precision greater than or equal to their specified bit precision. When your HLSL minimum precision shader code is used on hardware that implements HLSL minimum precision, you use less memory bandwidth and as a result you also use less system power.

You can query for the minimum precision support that the graphics driver provides by calling [ID3D11Device::CheckFeatureSupport](#) with the [D3D11_FEATURE_SHADER_MIN_PRECISION_SUPPORT](#) value. In this [ID3D11Device::CheckFeatureSupport](#) call, pass a pointer to a [D3D11_FEATURE_DATA_SHADER_MIN_PRECISION_SUPPORT](#) structure that [ID3D11Device::CheckFeatureSupport](#) fills with the minimum precision levels that the driver supports for the pixel shader stage and for other shader stages. The returned info just indicates that the graphics hardware can perform HLSL operations at a lower precision than the standard 32-bit float precision, but doesn't guarantee that the graphics hardware will actually run at a lower precision.

You don't need to author multiple shaders that do and don't use minimum precision. Instead, create shaders with minimum precision, and the minimum precision variables behave at full 32-bit precision if the graphics driver reports that it doesn't support any minimum precision. For more info about HLSL minimum precision, see [Using HLSL minimum precision](#).

HLSL minimum precision shaders don't work on operating systems earlier than Windows 8.

Specify user clip planes in HLSL on feature level 9 and higher

Starting with Windows 8, you can use the [clipplanes](#) function attribute in an HLSL [function declaration](#) rather than [SV_ClipDistance](#) to make your shader work on [feature level 9_x](#) as well as feature level 10 and higher. For more info, see [User clip planes on feature level 9 hardware](#).

Create larger constant buffers than a shader can access

Direct3D 11.1 lets you create constant buffers that are larger than the maximum constant buffer size that a shader can access (4096 32-bit*4-component constants – 64KB). Later, when you bind the buffers to the pipeline, for example, via [PSSetConstantBuffers](#) or [PSSetConstantBuffers1](#), you can specify a range of buffers that the shader can access that fits within the 4096 limit.

Direct3D 11.1 updates the [ID3D11Device::CreateBuffer](#) method for this feature.

Use logical operations in a render target

Direct3D 11.1 lets you use logical operations rather than blending in a render target. However, you can't mix logic operations with blending across multiple render targets.

This Direct3D 11.1 feature consists of the following API.

- [ID3D11Device1::CreateBlendState1](#) with [D3D11_LOGIC_OP](#)

Force the sample count to create a rasterizer state

Direct3D 11.1 lets you specify a force sample count when you create a rasterizer state.

This Direct3D 11.1 feature consists of the following API.

- [ID3D11Device1::CreateRasterizerState1](#)

NOTE

If you want to render with the sample count forced to 1 or greater, you must follow these guidelines:

- Don't bind depth-stencil views.
- Disable depth testing.
- Ensure the shader doesn't output depth.
- If you have any render-target views bound ([D3D11_BIND_RENDER_TARGET](#)) and you forced the sample count to greater than 1, ensure that every render target has only a single sample.
- Don't operate the shader at sample frequency. Therefore, [ID3D11ShaderReflection::IsSampleFrequencyShader](#) returns FALSE.

Otherwise, rendering behavior is undefined. For info about how to configure depth-stencil, see [Configuring Depth-Stencil Functionality](#).

Process video resources with shaders

Direct3D 11.1 lets you create views (SRV/RTV/UAV) to video resources so that Direct3D shaders can process those video resources. The format of an underlying video resource restricts the formats that the view can use. The [DXGI_FORMAT](#) enumeration contains new video resource format values. These [DXGI_FORMAT](#) values specify the valid view formats that you can create and how the Direct3D 11.1 runtime maps the view. You can create multiple views of different parts of the same surface, and depending on the format, the sizes of the views can differ from each other.

Direct3D 11.1 updates the following methods for this feature.

- [ID3D11Device::CreateShaderResourceView](#)
- [ID3D11Device::CreateRenderTargetView](#)
- [ID3D11Device::CreateUnorderedAccessView](#)

Extended support for shared Texture2D resources

Direct3D 11.1 guarantees that you can share Texture2D resources that you created with particular resource types and formats. To share Texture2D resources, use the [D3D11_RESOURCE_MISC_SHARED](#), [D3D11_RESOURCE_MISC_SHARED_KEYEDMUTEX](#), or a combination of the [D3D11_RESOURCE_MISC_SHARED_KEYEDMUTEX](#) and [D3D11_RESOURCE_MISC_SHARED_NTHANDLE](#) (new for Windows 8) flags when you create those resources.

Direct3D 11.1 guarantees that you can share Texture2D resources that you created with these [DXGI_FORMAT](#) values:

- [DXGI_FORMAT_R8G8B8A8_UNORM](#)
- [DXGI_FORMAT_R8G8B8A8_UNORM_SRGB](#)
- [DXGI_FORMAT_B8G8R8A8_UNORM](#)
- [DXGI_FORMAT_B8G8R8A8_UNORM_SRGB](#)
- [DXGI_FORMAT_B8G8R8X8_UNORM](#)
- [DXGI_FORMAT_B8G8R8X8_UNORM_SRGB](#)
- [DXGI_FORMAT_R10G10B10A2_UNORM](#)
- [DXGI_FORMAT_R16G16B16A16_FLOAT](#)

In addition, Direct3D 11.1 guarantees that you can share Texture2D resources that you created with a mipmap level of 1, array size of 1, bind flags of [D3D11_BIND_SHADER_RESOURCE](#) and [D3D11_BIND_RENDER_TARGET](#) combined, usage default ([D3D11_USAGE_DEFAULT](#)), and only these [D3D11_RESOURCE_MISC_FLAG](#) values:

- [D3D11_RESOURCE_MISC_SHARED](#)
- [D3D11_RESOURCE_MISC_SHARED_KEYEDMUTEX](#)
- [D3D11_RESOURCE_MISC_SHARED_NTHANDLE](#)
- [D3D11_RESOURCE_MISC_GDI_COMPATIBLE](#)

Direct3D 11.1 lets you share a greater variety of Texture2D resource types and formats. You can query for whether the graphics driver and hardware support extended Texture2D resource sharing by calling [ID3D11Device::CheckFeatureSupport](#) with the [D3D11_FEATURE_D3D11_OPTIONS](#) value. In this [ID3D11Device::CheckFeatureSupport](#) call, pass a pointer to a [D3D11_FEATURE_DATA_D3D11_OPTIONS](#) structure. [ID3D11Device::CheckFeatureSupport](#) sets the [ExtendedResourceSharing](#) member of [D3D11_FEATURE_DATA_D3D11_OPTIONS](#) to TRUE if the hardware and driver support extended Texture2D resource sharing.

If [ID3D11Device::CheckFeatureSupport](#) returns TRUE in [ExtendedResourceSharing](#), you can share resources that you created with these [DXGI_FORMAT](#) values:

- [DXGI_FORMAT_R32G32B32A32_TYPELESS](#)
- [DXGI_FORMAT_R32G32B32A32_FLOAT](#)
- [DXGI_FORMAT_R32G32B32A32_UINT](#)
- [DXGI_FORMAT_R32G32B32A32_SINT](#)
- [DXGI_FORMAT_R16G16B16A16_TYPELESS](#)
- [DXGI_FORMAT_R16G16B16A16_FLOAT](#)
- [DXGI_FORMAT_R16G16B16A16_UNORM](#)
- [DXGI_FORMAT_R16G16B16A16_UINT](#)
- [DXGI_FORMAT_R16G16B16A16_SNORM](#)
- [DXGI_FORMAT_R16G16B16A16_SINT](#)
- [DXGI_FORMAT_R10G10B10A2_UNORM](#)
- [DXGI_FORMAT_R10G10B10A2_UINT](#)
- [DXGI_FORMAT_R8G8B8A8_TYPELESS](#)
- [DXGI_FORMAT_R8G8B8A8_UNORM](#)
- [DXGI_FORMAT_R8G8B8A8_UNORM_SRGB](#)
- [DXGI_FORMAT_R8G8B8A8_UINT](#)
- [DXGI_FORMAT_R8G8B8A8_SNORM](#)
- [DXGI_FORMAT_R8G8B8A8_SINT](#)
- [DXGI_FORMAT_B8G8R8A8_TYPELESS](#)
- [DXGI_FORMAT_B8G8R8A8_UNORM](#)
- [DXGI_FORMAT_B8G8R8X8_UNORM](#)
- [DXGI_FORMAT_B8G8R8A8_UNORM_SRGB](#)
- [DXGI_FORMAT_B8G8R8X8_TYPELESS](#)
- [DXGI_FORMAT_B8G8R8X8_UNORM_SRGB](#)
- [DXGI_FORMAT_R32_TYPELESS](#)
- [DXGI_FORMAT_R32_FLOAT](#)
- [DXGI_FORMAT_R32_UINT](#)
- [DXGI_FORMAT_R32_SINT](#)
- [DXGI_FORMAT_R16_TYPELESS](#)

- DXGI_FORMAT_R16_FLOAT
- DXGI_FORMAT_R16_UNORM
- DXGI_FORMAT_R16_UINT
- DXGI_FORMAT_R16_SNORM
- DXGI_FORMAT_R16_SINT
- DXGI_FORMAT_R8_TYPELESS
- DXGI_FORMAT_R8_UNORM
- DXGI_FORMAT_R8_UINT
- DXGI_FORMAT_R8_SNORM
- DXGI_FORMAT_R8_SINT
- DXGI_FORMAT_A8_UNORM
- DXGI_FORMAT_AYUV
- DXGI_FORMAT_YUY2
- DXGI_FORMAT_NV12
- DXGI_FORMAT_NV11
- DXGI_FORMAT_P016
- DXGI_FORMAT_P010
- DXGI_FORMAT_Y216
- DXGI_FORMAT_Y210
- DXGI_FORMAT_Y416
- DXGI_FORMAT_Y410

If [ID3D11Device::CheckFeatureSupport](#) returns TRUE in **ExtendedResourceSharing**, you can share resources that you created with these features and flags:

- [D3D11_USAGE_DEFAULT](#)
- [D3D11_BIND_SHADER_RESOURCE](#)
- [D3D11_BIND_RENDER_TARGET](#)
- [D3D11_RESOURCE_MISC_GENERATE_MIPS](#)
- [D3D11_BIND_UNORDERED_ACCESS](#)
- Mipmap levels (one or more levels) in the 2D texture resources (specified in the **MipLevels** member of [D3D11_TEXTURE2D_DESC](#))
- Arrays of 2D texture resources (one or more textures) (specified in the **ArraySize** member of [D3D11_TEXTURE2D_DESC](#))
- [D3D11_BIND_DECODER](#)
- [D3D11_RESOURCE_MISC_RESTRICTED_CONTENT](#)
- [D3D11_RESOURCE_MISC_RESTRICT_SHARED_RESOURCE](#)
- [D3D11_RESOURCE_MISC_RESTRICT_SHARED_RESOURCE_DRIVER](#)

NOTE

When **ExtendedResourceSharing** is TRUE, you have more flexibility when you specify bind flags for sharing Texture2D resources. Not only does the graphics driver and hardware support more bind flags but also more possible combinations of bind flags. For example, you can specify just [D3D11_BIND_RENDER_TARGET](#) or no bind flags, and so on.

Even if [ID3D11Device::CheckFeatureSupport](#) returns TRUE in **ExtendedResourceSharing**, you still can't share resources that you created with these features and flags:

- [D3D11_BIND_DEPTH_STENCIL](#)
- 2D texture resources in multisample antialiasing (MSAA) mode (specified with [DXGI_SAMPLE_DESC](#))
- [D3D11_RESOURCE_MISC_RESOURCE_CLAMP](#)
- [D3D11_USAGE_IMMUTABLE](#), [D3D11_USAGE_DYNAMIC](#), or [D3D11_USAGE_STAGING](#) (that is, mappable)
- [D3D11_RESOURCE_MISC_TEXTURECUBE](#)

Change subresources with new copy options

Direct3D 11.1 lets you use new copy flags to copy and update subresources. When you copy a subresource, the source and destination resources can be identical and the source and destination regions can overlap.

This Direct3D 11.1 feature consists of the following API.

- [ID3D11DeviceContext1::CopySubresourceRegion1](#)
- [ID3D11DeviceContext1::UpdateSubresource1](#)

Discard resources and resource views

Direct3D 11.1 lets you discard resources and views of resources from the device context. This new functionality informs the GPU that existing content in resources or resource views are no longer needed.

This Direct3D 11.1 feature consists of the following API.

- [ID3D11DeviceContext1::DiscardResource](#)
- [ID3D11DeviceContext1::DiscardView](#)
- [ID3D11DeviceContext1::DiscardView1](#)

Support a larger number of UAVs

Direct3D 11.1 lets you use a larger number of UAVs when you bind resources to the [output-merger stage](#) and when you set an array of views for an unordered resource.

Direct3D 11.1 updates the following methods for this feature.

- [ID3D11DeviceContext::OMSetRenderTargetsAndUnorderedAccessViews](#)
- [ID3D11DeviceContext::CSSetUnorderedAccessViews](#)

Bind a subrange of a constant buffer to a shader

Direct3D 11.1 lets you bind a subrange of a constant buffer for a shader to access. You can supply a larger constant buffer and specify the subrange that the shader can use.

This Direct3D 11.1 feature consists of the following API.

- [ID3D11DeviceContext1::CSSetConstantBuffers1](#)
- [ID3D11DeviceContext1::DSSetConstantBuffers1](#)
- [ID3D11DeviceContext1::GSSetConstantBuffers1](#)
- [ID3D11DeviceContext1::HSSetConstantBuffers1](#)
- [ID3D11DeviceContext1::PSSetConstantBuffers1](#)
- [ID3D11DeviceContext1::VSSetConstantBuffers1](#)

Retrieve the subrange of a constant buffer that is bound to a shader

Direct3D 11.1 lets you retrieve the subrange of a constant buffer that is bound to a shader.

This Direct3D 11.1 feature consists of the following API.

- [ID3D11DeviceContext1::CSGetConstantBuffers1](#)
- [ID3D11DeviceContext1::DSGetConstantBuffers1](#)
- [ID3D11DeviceContext1::GSGetConstantBuffers1](#)
- [ID3D11DeviceContext1::HSGetConstantBuffers1](#)
- [ID3D11DeviceContext1::PSGetConstantBuffers1](#)
- [ID3D11DeviceContext1::VSGetConstantBuffers1](#)

Clear all or part of a resource view

Direct3D 11.1 lets you clear a resource view (RTV, UAV, or any video view of a Texture2D surface). You apply the same color value to all parts of the view.

This Direct3D 11.1 feature consists of the following API.

- [ID3D11DeviceContext1::ClearView](#)

Map SRVs of dynamic buffers with NO_OVERWRITE

Direct3D 11.1 lets you map shader resource views (SRV) of dynamic buffers with D3D11_MAP_WRITE_NO_OVERWRITE. The Direct3D 11 and earlier runtimes limited mapping to vertex or index buffers.

Direct3D 11.1 updates the [ID3D11DeviceContext::Map](#) method for this feature.

Use UAVs at every pipeline stage

Direct3D 11.1 lets you use the following shader model 5.0 instructions at all shader stages that were previously used in just pixel shaders and compute shaders.

- [dcl_uav_typed](#)
- [dcl_uav_raw](#)
- [dcl_uav_structured](#)
- [ld_raw](#)
- [ld_structured](#)
- [ld_uav_typed](#)
- [store_raw](#)
- [store_structured](#)
- [store_uav_typed](#)
- [sync_uglobal](#)
- All atomics and immediate atomics (for example, [atomic_and](#) and [imm_atomic_and](#))

Direct3D 11.1 updates the following methods for this feature.

- [ID3D11DeviceContext::CreateDomainShader](#)
- [ID3D11DeviceContext::CreateGeometryShader](#)
- [ID3D11DeviceContext::CreateGeometryShaderWithStreamOutput](#)
- [ID3D11DeviceContext::CreateHullShader](#)
- [ID3D11DeviceContext::CreateVertexShader](#)

These instructions existed in Direct3D 11.0 in ps_5_0 and cs_5_0. Because Direct3D 11.1 makes UAVs available at all shader stages, these instructions are available at all shader stages.

If you pass compiled shaders (VS/HS/DS/HS) that use any of these instructions to a create-shader function, like [CreateVertexShader](#), on devices that don't support UAVs at every stage (including existing drivers that are not implemented with this feature), the create-shader function fails. The create-shader function also fails if the shader tries to use a UAV slot beyond the set of UAV slots that the hardware supports.

The UAVs that are referenced by these instructions are shared across all pipeline stages. For example, a UAV that is bound at slot 0 at the [output-merger stage](#) is available at slot 0 to VS/HS/DS/GS/PS.

UAV accesses that you issue from within or across shader stages that execute within a given Draw*() or that you issue from the compute shader within a Dispatch*() aren't guaranteed to finish in the order in which you issued them. But all UAV accesses finish at the end of the Draw*() or Dispatch*().

Extended support for WARP devices

Direct3D 11.1 extends support for [WARP](#) devices, which you create by passing [D3D_DRIVER_TYPE_WARP](#) in the *DriverType* parameter of [D3D11CreateDevice](#).

Starting with Direct3D 11.1 WARP devices support:

- All Direct3D [feature levels](#) from 9.1 through to 11.1
- [Compute shaders](#) and [tessellation](#)
- Shared surfaces. That is, you can fully share surfaces between WARP devices, as well as between WARP devices in different processes.

WARP devices don't support these optional features:

- [doubles](#)
- [video encode or decode](#)
- [minimum precision shader support](#)

When you run a virtual machine (VM), Hyper-V, with your graphics processing unit (GPU) disabled, or without a display driver, you get a WARP device whose friendly name is "Microsoft Basic Display Adapter." This WARP device can run the full Windows experience, which includes DWM, Internet Explorer, and Windows Store apps. This WARP device also supports running legacy apps that use [DirectDraw](#) or running apps that use Direct3D 3 through Direct3D 11.1.

Use Direct3D in Session 0 processes

Starting with Windows 8 and Windows Server 2012, you can use most of the Direct3D APIs in Session 0 processes.

NOTE

These output, window, swap chain, and presentation-related APIs are not available in Session 0 processes because they don't apply to the Session 0 environment:

- [IDXGIFactory::CreateSwapChain](#)
- [IDXGIFactory::GetWindowAssociation](#)
- [IDXGIFactory::MakeWindowAssociation](#)
- [IDXGIAdapter::EnumOutputs](#)
- [ID3D11Debug::SetFeatureMask](#)
- [ID3D11Debug::SetPresentPerRenderOpDelay](#)
- [ID3D11Debug::SetSwapChain](#)
- [ID3D10Debug::SetFeatureMask](#)
- [ID3D10Debug::SetPresentPerRenderOpDelay](#)
- [ID3D10Debug::SetSwapChain](#)
- [D3D10CreateDeviceAndSwapChain](#)
- [D3D10CreateDeviceAndSwapChain1](#)
- [D3D11CreateDeviceAndSwapChain](#)

If you call one of the preceding APIs in a Session 0 process, it returns [DXGI_ERROR_NOT_CURRENTLY_AVAILABLE](#).

Support for shadow buffer on feature level 9

Use a subset of Direct3D 10_0+ shadow buffer features to implement shadow effects on [feature level 9_x](#). For info about what you need to do to implement shadow effects on feature level 9_x, see [Implementing shadow buffers for Direct3D feature level 9](#).

Related topics

[What's new in Direct3D 11](#)

Direct3D 11.2 Features

11/2/2020 • 4 minutes to read • [Edit Online](#)

The following functionality has been added in Direct3D 11.2, which is included with Windows 8.1, Windows RT 8.1, and Windows Server 2012 R2.

- [Tiled resources](#)
 - [Check tiled resources support](#)
- [Extended support for WARP devices](#)
- [Annotate graphics commands](#)
- [HLSL shader linking](#)
 - [Function linking graph \(FLG\)](#)
- [Inbox HLSL compiler](#)
- [Related topics](#)

Tiled resources

Direct3D 11.2 lets you create tiled resources that can be thought of as large logical resources that use small amounts of physical memory. Tiled resources are useful (for example) with terrain in games, and app UI.

Tiled resources are created by specifying the [D3D11_RESOURCE_MISC_TILED](#) flag. To work with tiled resource, use these API:

- [ID3D11Device2::GetResourceTiling](#)
- [ID3D11DeviceContext2::UpdateTiles](#)
- [ID3D11DeviceContext2::UpdateTileMappings](#)
- [ID3D11DeviceContext2::CopyTiles](#)
- [ID3D11DeviceContext2::CopyTileMappings](#)
- [ID3D11DeviceContext2::ResizeTilePool](#)
- [ID3D11DeviceContext2::TiledResourceBarrier](#)
- [D3D11_DEBUG_FEATURE_DISABLE_TILED_RESOURCE_MAPPING_TRACKING_AND_VALIDATION](#) flag with [ID3D11Debug::SetFeatureMask](#)

For more info about tiled resources, see [Tiled resources](#).

Check tiled resources support

Before you use tiled resources, you need to find out if the device supports tiled resources. Here's how you check for support for tiled resources:

```

HRESULT hr = D3D11CreateDevice(
    nullptr, // Specify nullptr to use the default adapter.
    D3D_DRIVER_TYPE_HARDWARE, // Create a device using the hardware graphics driver.
    0, // Should be 0 unless the driver is D3D_DRIVER_TYPE_SOFTWARE.
    creationFlags, // Set debug and Direct2D compatibility flags.
    featureLevels, // List of feature levels this app can support.
    ARRSIZE(featureLevels), // Size of the list above.
    D3D11_SDK_VERSION, // Always set this to D3D11_SDK_VERSION for Windows Store apps.
    &device, // Returns the Direct3D device created.
    &m_d3dFeatureLevel, // Returns feature level of device created.
    &context // Returns the device immediate context.
);

if (SUCCEEDED(hr))
{
    D3D11_FEATURE_DATA_D3D11_OPTIONS1 featureData;
    DX::ThrowIfFailed(
        device->CheckFeatureSupport(D3D11_FEATURE_D3D11_OPTIONS1, &featureData, sizeof(featureData))
    );

    m_tiledResourcesTier = featureData.TiledResourcesTier;
}

```

Extended support for WARP devices

Direct3D 11.2 extends support for [WARP](#) devices, which you create by passing [D3D_DRIVER_TYPE_WARP](#) in the *DriverType* parameter of [D3D11CreateDevice](#). The WARP software renderer in Direct3D 11.2 adds full support for Direct3D [feature level 11_1](#), including [tiled resources](#), [IDXGIDevice3::Trim](#), shared BCn surfaces, minblend, and map default. [Double](#) support in HLSL shaders has also been enabled along with support for 16x MSAA.

Annotate graphics commands

Direct3D 11.2 lets you annotate graphics commands with these API:

- [ID3D11DeviceContext2::IsAnnotationEnabled](#)
- [ID3D11DeviceContext2::BeginEventInt](#)
- [ID3D11DeviceContext2::SetMarkerInt](#)
- [ID3D11DeviceContext2::EndEvent](#)

HLSL shader linking

Windows 8.1 adds separate compilation and linking of HLSL shaders, which allows graphics programmers to create precompiled HLSL functions, package them into libraries, and link them into full shaders at run-time. This is essentially an equivalent of C/C++ separate compilation, libraries, and linking, and it allows programmers to compose precompiled HLSL code when more information becomes available to finalize the computation. For more info about how to use shader linking, see [Using shader linking](#).

Complete these steps to create a final shader using dynamic linkage at run time.

To create and use shader linking

1. Create a [ID3D11Linker](#) linker object, which represents a linking context. A single context can't be used to produce multiple shaders; a linking context is used to produce a single shader and then the linking context is thrown away.
2. Use [D3DLoadModule](#) to load and set libraries from their library blobs.
3. Use [D3DLoadModule](#) to load and set an entry shader blob, or create an [FLG](#) shader.

4. Use [ID3D11Module::CreateInstance](#) to create [ID3D11ModuleInstance](#) objects, then call functions on these objects to rebind resources to their final slots.
5. Add the libraries to the linker, then call [ID3D11Linker::Link](#) to produce final shader byte code that can then be loaded and used in the runtime just like a fully precompiled and linked shader.

Function linking graph (FLG)

Windows 8.1 also adds the Function Linking Graph (FLG). You can use FLG to construct shaders that consist of a sequence of precompiled function invocations that pass values to each other. When using the FLG, there is no need to write HLSL and invoke the HLSL compiler. Instead, the shader structure is specified programmatically using C++ API calls. FLG nodes represent input and output signatures and invocations of precompiled library functions. The order of registering the function-call nodes defines the sequence of invocations. The input signature node must be specified first, while the output signature node must be specified last. FLG edges define how values are passed from one node to another. The data types of passed values must be the same; there is no implicit type conversion. Shape and swizzling rules follow the HLSL behavior and values can only be passed forward in this sequence. For info on the FLG API, see [ID3D11FunctionLinkingGraph](#).

Inbox HLSL compiler

The HLSL compiler is now inbox on Windows 8.1 and later. Now, most APIs for shader programming can be used in Windows Store apps that are built for Windows 8.1 and later. Many APIs for shader programming couldn't be used in Windows Store apps that were built for Windows 8; the reference pages for these APIs were marked with a note. But some shader APIs (for example, [D3DCompileFromFile](#)) can still only be used to develop Windows Store apps, and not in apps that you submit to the Windows Store; the reference pages for these APIs are still marked with a note.

Related topics

[What's new in Direct3D 11](#)

Direct3D 11.3 Features

2/4/2021 • 2 minutes to read • [Edit Online](#)

The following sections describe the functionality has been added in Direct3D 11.3. These features are also available in Direct3D 12.

In this section

TOPIC	DESCRIPTION
Conservative Rasterization	Conservative rasterization adds some certainty to pixel rendering, which is helpful in particular to collision detection algorithms.
Default Texture Mapping	The use of default texture mapping reduces copying and memory usage while sharing image data between the GPU and the CPU. However, it should only be used in specific situations. The standard swizzle layout avoids copying or swizzling data in multiple layouts.
Rasterizer Order Views	Rasterizer ordered views (ROVs) allow pixel shader code to mark UAV bindings with a declaration that alters the normal requirements for the order of graphics pipeline results for UAVs. This enables Order Independent Transparency (OIT) algorithms to work, which give much better rendering results when multiple transparent objects are in line with each other in a view.
Shader Specified Stencil Reference Value	Enabling pixel shaders to output the Stencil Reference Value, rather than using the API-specified one, enables a very fine granular control over stencil operations.
Typed Unordered Access View Loads	Unordered Access View (UAV) Typed Load is the ability for a shader to read from a UAV with a specific DXGI_FORMAT.
Unified Memory Architecture	Querying for whether Unified Memory Architecture (UMA) is supported can help determine how to handle some resources.
Volume Tiled Resources	Volume (3D) textures can be used as tiled resources, noting that tile resolution is three-dimensional.

Related topics

[What's new in Direct3D 11](#)

Direct3D 11.3 Conservative Rasterization

11/2/2020 • 3 minutes to read • [Edit Online](#)

Conservative rasterization adds some certainty to pixel rendering, which is helpful in particular to collision detection algorithms.

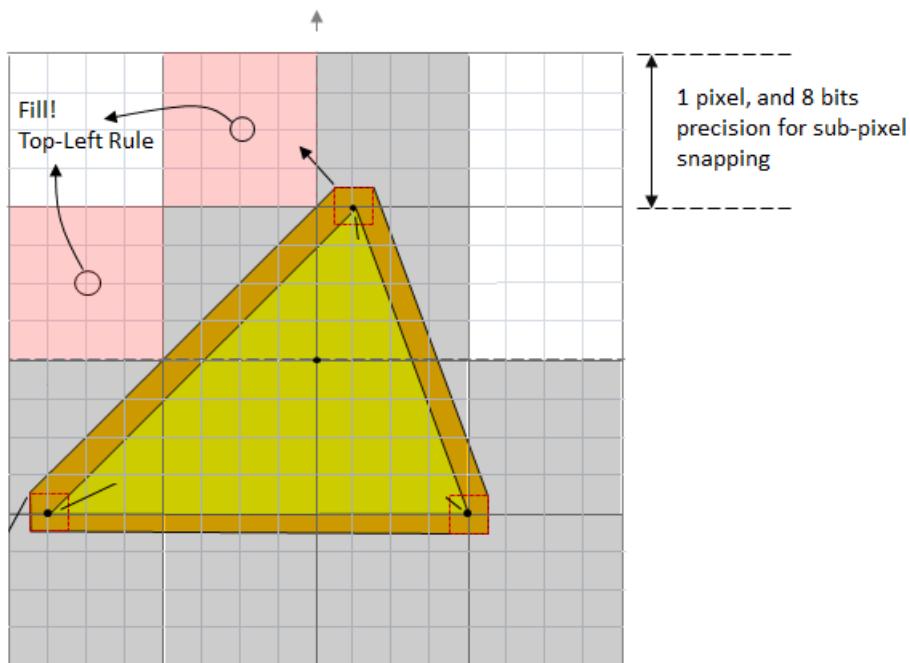
- [Overview](#)
- [Interactions with the pipeline](#)
- [Implementation details](#)
- [API summary](#)
- [Related topics](#)

Overview

Conservative rasterization means that all pixels that are at least partially covered by a rendered primitive are rasterized, which means that the pixel shader is invoked. Normal behavior is sampling, which is not used if conservative rasterization is enabled.

Conservative rasterization is useful in a number of situations, including for certainty in collision detection, occlusion culling, and visibility detection.

For example, the following figure shows a green triangle rendered using conservative rasterization. The brown area is known as an "uncertainty region" - a region where rounding errors and other issues add some uncertainty to the exact dimensions of the triangle. The red triangles at each vertex show how the uncertainty region is calculated. The large gray squares show the pixels that will be rendered. The pink squares show pixels rendered using the "top-left rule", which comes into play as the edge of the triangle crosses the edge of the pixels. There can be false positives (pixels set that should not have been) which the system will normally but not always cull.



Interactions with the pipeline

For many details on how conservative rasterization interacts with the graphics pipeline, refer to [D3D12 Conservative rasterization](#).

Implementation details

The type of rasterization supported in Direct3D 12 is sometimes referred to as "overestimated conservative rasterization". There is also the concept of "underestimated conservative rasterization", which means that only pixels that are fully covered by a rendered primitive are rasterized. Underestimated conservative rasterization information is available through the pixel shader through the use of input coverage data, and only overestimated conservative rasterization is available as a rasterizing mode.

If any part of a primitive overlaps a pixel, then that pixel is considered covered and is then rasterized. When an edge or corner of a primitive falls along the edge or corner of a pixel, the application of the "top-left rule" is implementation-specific. However, for implementations that support degenerate triangles, a degenerate triangle along an edge or corner must cover at least one pixel.

Conservative rasterization implementations can vary on different hardware, and do produce false positives, meaning that they can incorrectly decide that pixels are covered. This can occur because of implementation-specific details like primitive growing or snapping errors inherent in the fixed-point vertex coordinates used in rasterization. The reason false positives (with respect to fixed point vertex coordinates) are valid is because some amount of false positives are needed to allow an implementation to do coverage evaluation against post-snapped vertices (i.e. vertex coordinates that have been converted from floating point to the 16.8 fixed-point used in the rasterizer), but honor the coverage produced by the original floating point vertex coordinates.

Conservative rasterization implementations do not produce false negatives with respect to the floating-point vertex coordinates for non-degenerate post-snap primitives: if any part of a primitive overlaps any part of a pixel, then that pixel is rasterized.

Triangles that are degenerate (duplicate indices in an index buffer or collinear in 3D), or become degenerate after fixed-point conversion (collinear vertices in the rasterizer), may or may not be culled; both are valid behaviors. Degenerate triangles must be considered back facing, so if a specific behavior is required by an application, it can use back-face culling or test for front facing. Degenerate triangles use the values assigned to Vertex 0 for all interpolated values.

There are three tiers of hardware support, in addition to the possibility that the hardware does not support this feature.

- Tier 1 supports 1/2 pixel uncertainty regions, and no post-snap degenerates. This is good for tiled rendering, a texture atlas, light map generation and sub-pixel shadow maps.
- Tier 2 adds post-snap degenerates, and 1/256 uncertainty regions. It also adds support for CPU-based algorithm acceleration (such as voxelization).
- Tier 3 adds 1/512 uncertainty regions, inner input coverage and supports occlusion culling. The input coverage adds the new value `sv_InnerCoverage` to High Level Shading Language (HLSL). This is a 32-bit scalar integer that can be specified on input to a pixel shader, and represents the underestimated conservative rasterization information (that is, whether a pixel is guaranteed-to-be-fully covered).

API summary

The following methods, structures, enums, and helper classes reference conservative rasterization:

- [D3D11_RASTERIZER_DESC2](#) : structure holding the rasterizer description, noting the CD3D12_RASTERIZER_DESC2 helper class for creating rasterizer descriptions.
- [D3D11_CONSERVATIVE_RASTERIZATION_MODE](#) : enum values for the mode (on or off).
- [D3D11_FEATURE_DATA_D3D11_OPTIONS2](#) : structure holding the tier of support.
- [D3D11_CONSERVATIVE_RASTERIZATION_TIER](#) : enum values for each tier of support by the hardware.

- [ID3D11Device::CheckFeatureSupport](#) : method to access the supported features.

Related topics

- [Direct3D 11.3 Features](#)

Default Texture Mapping

2/22/2020 • 2 minutes to read • [Edit Online](#)

The use of default texture mapping reduces copying and memory usage while sharing image data between the GPU and the CPU. However, it should only be used in specific situations. The standard swizzle layout avoids copying or swizzling data in multiple layouts.

- [Overview](#)
- [D3D11.3 APIs](#)
- [Related topics](#)

Overview

Mapping default textures should not be the first choice for developers. Developers should code in a discrete-GPU friendly way first (that is, do not have any CPU access for the majority of textures and upload with [CopySubresourceRegion1](#)). But, for certain cases, the CPU and GPU may interact so frequently on the same data, that mapping default textures becomes helpful to save power, or to speed up a particular design on particular adapters or architectures. Applications should detect these cases and optimize out the unnecessary copies.

In D3D11.3, textures created with `D3D11_TEXTURE_LAYOUT_UNDEFINED` (one member of the [D3D11_TEXTURE_LAYOUT](#) enum) and no CPU access are the most efficient for frequent GPU rendering and sampling. When performance testing, those textures should be compared against `D3D11_TEXTURE_LAYOUT_UNDEFINED` with CPU access, and `D3D11_TEXTURE_LAYOUT_64K_STANDARD_SWIZZLE` with CPU access, and `D3D11_TEXTURE_LAYOUT_ROW_MAJOR` for cross-adapter support.

Using `D3D11_TEXTURE_LAYOUT_UNDEFINED` with CPU access enables the methods [WriteToSubresource](#), [ReadFromSubresource](#), [Map](#) (precluding application access to pointer), and [Unmap](#); but can sacrifice efficiency of GPU access. Using `D3D11_TEXTURE_LAYOUT_64K_STANDARD_SWIZZLE` with CPU access enables [WriteToSubresource](#), [ReadFromSubresource](#), [Map](#) (which returns a valid pointer to application), and [Unmap](#). It can also sacrifice efficiency of GPU access more than `D3D11_TEXTURE_LAYOUT_UNDEFINED` with CPU access.

In general, applications should create the majority of textures as GPU-only-accessible, and with `D3D11_TEXTURE_LAYOUT_UNDEFINED`.

Previous to the mapping default textures feature, there was only one standardized layout for multi-dimensional data: "linear", also known as "row-major". Applications should avoid `USAGE_STAGING` and `USAGE_DYNAMIC` textures when map default is available. The `USAGE_STAGING` and `USAGE_DYNAMIC` textures use the linear layout.

D3D11.3 (and D3D12) introduce a standard multi-dimensional data layout. This is done to enable multiple processing units to operate on the same data without copying the data or swizzling the data between multiple layouts. A standardized layout enables efficiency gains through network effects and allows algorithms to make short-cuts assuming a particular pattern.

Note though that this standard swizzle is a hardware feature, and may not be supported by all GPUs.

D3D11.3 APIs

Unlike D3D12, D3D11.3 does not supports texture mapping by default, so you need to query

[D3D11_FEATURE_DATA_D3D11_OPTIONS2](#). Standard swizzle will also need to be queried for with a call to [ID3D11Device::CheckFeatureSupport](#) and checking the `StandardSwizzle64KBSupported` field of [D3D11_FEATURE_DATA_D3D11_OPTIONS2](#).

The following APIs reference texture mapping:

Enums

- [D3D11_TEXTURE_LAYOUT](#) : controls the swizzle pattern of default textures and enable map support on default textures.
- [D3D11_FEATURE](#) : references [D3D11_FEATURE_DATA_D3D11_OPTIONS2](#).
- [D3D11_TILE_COPY_FLAG](#) : contains flags to copy swizzled tiled resources to and from linear buffers.

Structures

- [D3D11_TEXTURE2D_DESC1](#) : describes a 2D texture. Note the CD3D11_TEXTURE2D_DESC1 helper structure.
- [D3D11_TEXTURE3D_DESC1](#) : describes a 3D texture. Note the CD3D11_TEXTURE3D_DESC1 helper structure.

Methods

- [ID3D11Device3::CreateTexture2D1](#) : creates an array of 2D textures.
- [ID3D11Device3::CreateTexture3D1](#) : creates a single 3D texture.
- [ID3D11Device3::WriteToSubresource](#) : copies data into a D3D11_USAGE_DEFAULT texture which was mapped using [Map](#).
- [ID3D11Device3::ReadFromSubresource](#) : copies data from a D3D11_USAGE_DEFAULT texture which was mapped using [Map](#).
- [ID3D11DeviceContext::Map](#) : gets a pointer to the data contained in a subresource, and denies the GPU access to that subresource.
- [ID3D11DeviceContext::Unmap](#) : invalidates the pointer to a resource and reenables the GPU's access to that resource.
- [ID3D11Texture2D1::GetDesc1](#) : gets the properties of a 2D texture resource.
- [ID3D11Texture3D1::GetDesc1](#) : gets the properties of a 3D texture resource.

Related topics

[Direct3D 11.3 Features](#)

Rasterizer Order Views

11/2/2020 • 2 minutes to read • [Edit Online](#)

Rasterizer ordered views (ROVs) allow pixel shader code to mark UAV bindings with a declaration that alters the normal requirements for the order of graphics pipeline results for UAVs. This enables Order Independent Transparency (OIT) algorithms to work, which give much better rendering results when multiple transparent objects are in line with each other in a view.

- [Overview](#)
- [Implementation details](#)
- [API summary](#)
- [Related topics](#)

Overview

Standard graphics pipelines may have trouble correctly compositing together multiple textures that contain transparency. Objects such as wire fences, smoke, fire, vegetation, and colored glass use transparency to get the desired effect. Problems arise when multiple textures that contain transparency are in line with each other (smoke in front of a fence in front of a glass building containing vegetation, as an example). Rasterizer ordered views (ROVs) enable the underlying OIT algorithms to use features of the hardware to try to resolve the transparency order correctly. Transparency is handled by the pixel shader.

Rasterizer ordered views (ROVs) allow pixel shader code to mark UAV bindings with a declaration that alters the normal requirements for the order of graphics pipeline results for UAVs.

ROVs guarantee the order of UAV accesses for any pair of overlapping pixel shader invocations. In this case “overlapping” means that the invocations are generated by the same draw calls and share the same pixel coordinate when in pixel-frequency execution mode, and the same pixel and sample coordinate in sample-frequency mode.

The order in which overlapping ROV accesses of pixel shader invocations are executed is identical to the order in which the geometry is submitted. This means that, for overlapping pixel shader invocations, ROV writes performed by a pixel shader invocation must be available to be read by a subsequent invocation and must not affect reads by a previous invocation. ROV reads performed by a pixel shader invocation must reflect writes by a previous invocation and must not reflect writes by a subsequent invocation. This is important for UAVs because they are explicitly omitted from the output-invariance guarantees normally set by the fixed order of graphics pipeline results.

Implementation details

Rasterizer ordered views (ROVs) are declared with the following new High Level Shader Language (HLSL) objects, and are only available to the pixel shader:

- `RasterizerOrderedBuffer`
- `RasterizerOrderedByteAddressBuffer`
- `RasterizerOrderedStructuredBuffer`
- `RasterizerOrderedTexture1D`
- `RasterizerOrderedTexture1DArray`
- `RasterizerOrderedTexture2D`
- `RasterizerOrderedTexture2DArray`

- `RasterizerOrderedTexture3D`

Use these objects in the same manner as other UAV objects (such as `RWBuffer` etc.).

API summary

ROVs are an HLSL-only construct that applies different behavior semantics to UAVs. All APIs relevant to UAVs are also relevant to ROVs. Note that the following method, structures, and helper class reference the rasterizer:

- [D3D11_RASTERIZER_DESC2](#) : structure holding the rasterizer description, noting the `CD3D12_RASTERIZER_DESC2` helper class for creating rasterizer descriptions.
- [D3D11_FEATURE_DATA_D3D11_OPTIONS2](#) : structure holding the boolean `ROVsSupported`, indicating the support.
- [ID3D11Device::CheckFeatureSupport](#) : method to access the supported features.

Related topics

[Direct3D 11.3 Features](#)

[Shader Model 5.1](#)

Shader Specified Stencil Reference Value (Direct3D 11 Graphics)

2/4/2021 • 2 minutes to read • [Edit Online](#)

Enabling pixel shaders to output the Stencil Reference Value, rather than using the API-specified one, enables a very fine granular control over stencil operations.

The shader specified value replaces the API-specified *Stencil Reference Value* for that invocation, which means the change affects both the stencil test, and when stencil op D3D11_STENCIL_OP_REPLACE (one member of [D3D11_STENCIL_OP](#)) is used to write the reference value to the stencil buffer.

In earlier versions of D3D11, the Stencil Reference Value can only be specified by the [ID3D11DeviceContext::OMSetDepthStencilState](#) method. This means that this value can only be defined on a per-draw granularity. This D3D11.3 feature enables developers to read and use the Stencil Reference Value ([sv_StencilRef](#)) that is output from a pixel shader, meaning that it can be specified on a per-pixel or per-sample granularity.

This feature is optional in D3D11.3. To test for its support, check the [PSSpecifiedStencilRefSupported](#) boolean field of [D3D11_FEATURE_DATA_D3D11_OPTIONS2](#) using [ID3D11Device::CheckFeatureSupport](#)

Here is an example of the use of [sv_StencilRef](#) in a pixel shader:

```
uint main2(float4 c : COORD) : SV_StencilRef
{
    return uint(c.x);
}
```

Related topics

[Direct3D 11.3 Features](#)

[Shader Model 5.1](#)

Typed Unordered Access View Loads

11/2/2020 • 2 minutes to read • [Edit Online](#)

Unordered Access View (UAV) Typed Load is the ability for a shader to read from a UAV with a specific DXGI_FORMAT.

- [Overview](#)
- [Supported formats and API calls](#)
- [Using typed UAV loads from HLSL](#)
- [Related topics](#)

Overview

An unordered access view (UAV) is a view of an unordered access resource (which can include buffers, textures, and texture arrays, though without multi-sampling). A UAV allows temporally unordered read/write access from multiple threads. This means that this resource type can be read/written simultaneously by multiple threads without generating memory conflicts. This simultaneous access is handled through the use of [Atomic Functions](#).

D3D12 and D3D11.3 expands on the list of formats that can be used with typed UAV loads.

Supported formats and API calls

Previously, the following three formats supported typed UAV loads, and were required for D3D11.0 hardware. They are supported for all D3D11.3 and D3D12 hardware.

- R32_FLOAT
- R32_UINT
- R32_SINT

The following formats are supported as a set on D3D12 or D3D11.3 hardware, so if any one is supported, all are supported.

- R32G32B32A32_FLOAT
- R32G32B32A32_UINT
- R32G32B32A32_SINT
- R16G16B16A16_FLOAT
- R16G16B16A16_UINT
- R16G16B16A16_SINT
- R8G8B8A8_UNORM
- R8G8B8A8_UINT
- R8G8B8A8_SINT
- R16_FLOAT
- R16_UINT
- R16_SINT
- R8_UNORM
- R8_UINT
- R8_SINT

The following formats are optionally and individually supported on D3D12 and D3D11.3 hardware, so a single

query would need to be made on each format to test for support.

- R16G16B16A16_UNORM
- R16G16B16A16_SNORM
- R32G32_FLOAT
- R32G32_UINT
- R32G32_SINT
- R10G10B10A2_UNORM
- R10G10B10A2_UINT
- R11G11B10_FLOAT
- R8G8B8A8_SNORM
- R16G16_FLOAT
- R16G16_UNORM
- R16G16_UINT
- R16G16_SNORM
- R16G16_SINT
- R8G8_UNORM
- R8G8_UINT
- R8G8_SNORM
- 8G8_SINT
- R16_UNORM
- R16_SNORM
- R8_SNORM
- A8_UNORM
- B5G6R5_UNORM
- B5G5R5A1_UNORM
- B4G4R4A4_UNORM

To determine the support for any additional formats, call [ID3D11Device::CheckFeatureSupport](#) with the

[D3D11_FEATURE_DATA_D3D11_OPTIONS2](#) structure as the first parameter. The

`TypedUAVLoadAdditionalFormats` bit will be set if the "supported as a set" list above is supported. Make a second call to [CheckFeatureSupport](#), using a [D3D11_FEATURE_DATA_FORMAT_SUPPORT2](#) structure (checking the returned structure against the `D3D12_FORMAT_SUPPORT2_UAV_TYPED_LOAD` member of the [D3D11_FORMAT_SUPPORT2](#) enum) to determine support in the list of optionally supported formats above, for example:

```

D3D11_FEATURE_DATA_D3D11_OPTIONS2 FeatureData;
ZeroMemory(&FeatureData, sizeof(FeatureData));
HRESULT hr = pDevice->CheckFeatureSupport(D3D11_FEATURE_D3D11_OPTIONS2, &FeatureData, sizeof(FeatureData));
if (SUCCEEDED(hr))
{
    // TypedUAVLoadAdditionalFormats contains a Boolean that tells you whether the feature is supported or
    // not
    if (FeatureData.TypedUAVLoadAdditionalFormats)
    {
        // Can assume "all-or-nothing" subset is supported (e.g. R32G32B32A32_FLOAT)
        // Can not assume other formats are supported, so we check:
        D3D11_FEATURE_DATA_FORMAT_SUPPORT2 FormatSupport;
        ZeroMemory(&FormatSupport, sizeof(FormatSupport));
        FormatSupport.InFormat = DXGI_FORMAT_R32G32_FLOAT;
        hr = pDevice->CheckFeatureSupport(D3D11_FEATURE_FORMAT_SUPPORT2, &FormatSupport,
        sizeof(FormatSupport));
        if (SUCCEEDED(hr) && (FormatSupport.OutFormatSupport2 & D3D11_FORMAT_SUPPORT2_UAV_TYPED_LOAD) != 0)
        {
            // DXGI_FORMAT_R32G32_FLOAT supports UAV Typed Load!
        }
    }
}

```

Using typed UAV loads from HLSL

For typed UAVs, the HLSL flag is D3D_SHADER_REQUIRES_TYPED_UAV_LOAD_ADDITIONAL_FORMATS.

Here is example shader code to process a typed UAV load:

```

RWTexture2D<float4> uav1;
uint2 coord;
float4 main() : SV_Target
{
    return uav1.Load(coord);
}

```

Related topics

[Direct3D 11.3 Features](#)

[Shader Model 5.1](#)

Unified Memory Architecture

2/22/2020 • 2 minutes to read • [Edit Online](#)

Querying for whether Unified Memory Architecture (UMA) is supported can help determine how to handle some resources.

A boolean, set by the driver, can be read from the [D3D11_FEATURE_DATA_D3D11_OPTIONS2](#) structure to determine if the hardware supports UMA.

Applications running on UMA may want to have more resources with CPU access enabled than if it is not available. UMA enables applications to avoid copying resource data around, instead of staying efficient solely for non-UMA graphics adapters. [Direct3D 11.3 Features](#)

Related topics

[Direct3D 11.3 Features](#)

Volume Tiled Resources

2/22/2020 • 2 minutes to read • [Edit Online](#)

Volume (3D) textures can be used as tiled resources, noting that tile resolution is three-dimensional.

- [Overview](#)
- [D3D11.3 Tiled Resource APIs](#)
- [Related topics](#)

Overview

Tiled resources decouple a D3D Resource object from its backing memory (resources in the past had a 1:1 relationship with their backing memory). This allows for a variety of interesting scenarios such as streaming in texture data and reusing or reducing memory usage

2D texture tiled resources are supported in D3D11.2. D3D12 and D3D11.3 add support for 3D tiled textures.

The typical resource dimensions used in tiling are 4 x 4 tiles for 2D textures, and 4 x 4 x 4 tiles for 3D textures.

Bits/pixel (1 sample/pixel)	Tile dimensions (pixels, w x h x d)
8	64x32x32
16	32x32x32
32	32x32x16
64	32x16x16
128	16x16x16
BC 1,4	128x64x16
BC 2,3,5,6,7	64x64x16

Note the following formats are not supported with tiled resources: 96bpp formats, video formats, R1_UNORM, R8G8_B8G8_UNORM, R8R8_G8B8_UNORM.

In the diagrams below dark gray represents NULL tiles.

- [Texture 3D Tiled Resource default mapping \(most detailed mip\)](#)
- [Texture 3D Tiled Resource default mapping \(second most detailed mip\)](#)
- [Texture 3D Tiled Resource \(most detailed mip\)](#)
- [Texture 3D Tiled Resource \(second most detailed mip\)](#)
- [Texture 3D Tiled Resource \(Single Tile\)](#)
- [Texture 3D Tiled Resource \(Uniform Box\)](#)

Texture 3D Tiled Resource default mapping (most detailed mip)

Slice 0				Slice 1				Slice 2				Slice 3			
A	B	C	D	Q	R	S	T	g	h	i	j	w	x	y	z
E	F	G	H	U	V	W	X	k	l	m	n	0	1	2	3
I	J	K	L	Y	Z	a	b	o	p	q	r	4	5	6	7
M	N	O	P	c	d	e	f	s	t	u	v	8	9	+	/

Tile Pool

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
Q	R	S	T	U	V	W	X	Y	Z	a	b	c	d	e	f
g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v
w	x	y	z	0	1	2	3	4	5	6	7	8	9	+	/

Texture 3D Tiled Resource default mapping (second most detailed mip)

Slice 0		Slice 1	
A	B	E	F
C	D	G	H

Tile Pool

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
Q	R	S	T	U	V	W	X	Y	Z	a	b	c	d	e	f
g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v
w	x	y	z	0	1	2	3	4	5	6	7	8	9	+	/

Texture 3D Tiled Resource (most detailed mip)

The following code sets up a 3D tiled resource at the most detailed mip.

```

D3D11_TILED_RESOURCE_COORDINATE trCoord;
trCoord.X = 1;
trCoord.Y = 0;
trCoord.Z = 0;
trCoord.Subresource = 0;

D3D11_TILE_REGION_SIZE trSize;
trSize.bUseBox = false;
trSize.NumTiles = 63;

```

Slice 0				Slice 1				Slice 2				Slice 3			
	B	C	D	Q	R			v	w			m	n	o	p
E	F	G	H	f	g	h									
I	J	K	L		Z										
M	N	O	P			u						l			

Tile Pool

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
Q	R	S	T	U	V	W	X	Y	Z	a	b	c	d	e	f
g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v
w	x	y	z	0	1	2	3	4	5	6	7	8	9	+	/

Texture 3D Tiled Resource (second most detailed mip)

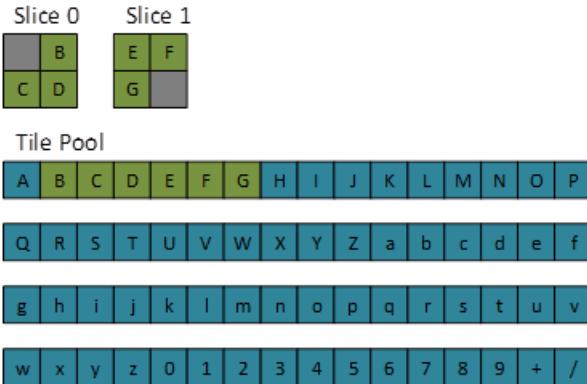
The following code sets up a 3D tiled resource, and the second most detailed mip:

```

D3D11_TILED_RESOURCE_COORDINATE trCoord;
trCoord.X = 1;
trCoord.Y = 0;
trCoord.Z = 0;
trCoord.Subresource = 1;

D3D11_TILE_REGION_SIZE trSize;
trSize.bUseBox = false;
trSize.NumTiles = 6;

```



Texture 3D Tiled Resource (Single Tile)

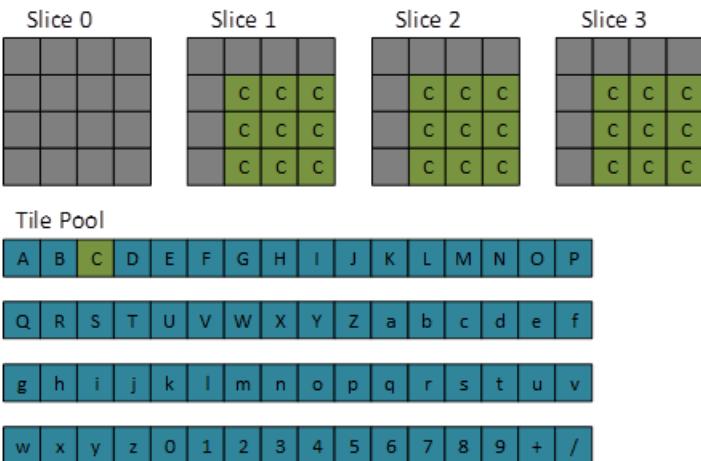
The following code sets up a Single Tile resource:

```

D3D11_TILED_RESOURCE_COORDINATE trCoord;
trCoord.X = 1;
trCoord.Y = 1;
trCoord.Z = 1;
trCoord.Subresource = 0;

D3D11_TILE_REGION_SIZE trSize;
trSize.bUseBox = true;
trSize.NumTiles = 27;
trSize.Width = 3;
trSize.Height = 3;
trSize.Depth = 3;

```



Texture 3D Tiled Resource (Uniform Box)

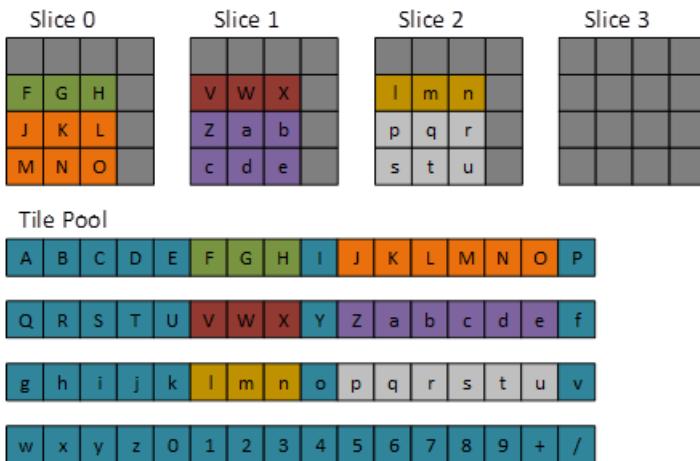
The following code sets up a Uniform Box tiled resource (note the statement `trSize.bUseBox = true;`) :

```

D3D11_TILED_RESOURCE_COORDINATE trCoord;
trCoord.X = 0;
trCoord.Y = 1;
trCoord.Z = 0;
trCoord.Subresource = 0;

D3D11_TILE_REGION_SIZE trSize;
trSize.bUseBox = true;
trSize.NumTiles = 27;
trSize.Width = 3;
trSize.Height = 3;
trSize.Depth = 3;

```



D3D11.3 Tiled Resource APIs

The same API calls are used for both 2D and 3D tiled resources:

Enums

- [D3D11_TILED_RESOURCES_TIER](#) : determines the level of tiled resource support.
- [D3D11_FORMAT_SUPPORT2](#) : used to test for tiled resource support.
- [D3D11_CHECK_MULTISAMPLE_QUALITY_LEVELS_FLAG](#) : determines tiled resource support in a multi-sampling resource.
- [D3D11_TILE_COPY_FLAGS](#) : holds flags for copying to and from swizzled tiled resources and linear buffers.

Structures

- [D3D11_TILED_RESOURCE_COORDINATE](#) : holds the x, y, and z co-ordinate, and subresource reference. Note there is a helper class: CD3D11_TILED_RESOURCE_COORDINATE.
- [D3D11_TILE_REGION_SIZE](#) : specifies the size, and number of tiles, of the tiled region.
- [D3D11_TILE_SHAPE](#) : the tile shape as a width, height and depth in texels.
- [D3D11_FEATURE_DATA_D3D11_OPTIONS1](#): holds the supported tile resource tier level.

Methods

- [ID3D11Device::CheckFeatureSupport](#) : used to determine what features, and at what tier, are supported by the current hardware.
- [ID3D11DeviceContext2::CopyTiles](#) : copies tiles from buffer to tiled resource or vice versa.
- [ID3D11DeviceContext2::UpdateTileMappings](#) : updates mappings of tile locations in tiled resources to memory locations in a tile pool.
- [ID3D11DeviceContext2::CopyTileMappings](#) : copies mappings from a source tiled resource to a

destination tiled resource.

- [ID3D11DeviceContext2::GetResourceTiling](#) : gets info about how a tiled resource is broken into tiles.

Related topics

[Direct3D 11.3 Features](#)

Direct3D 11.4 Features

11/2/2020 • 2 minutes to read • [Edit Online](#)

The following functionality has been added in Direct3D 11.4.

Also see [Where is the DirectX SDK?](#).

Direct3D device removal

The [RegisterDeviceRemovedEvent](#), and [UnregisterDeviceRemoved](#) methods are supported by a new interface, [ID3D11Device4](#), to support receiving an asynchronous event notification when a Direct3D device has been removed.

Multithreaded protection

To ensure that graphics commands in particular are executed in a specific order, the [ID3D11Multithread](#) interface has methods to turn multithread protection on and off, and methods to enter and leave critical code requiring this protection.

Fences for multi-device synchronization and interop with Direct3D 12

The [ID3D11Fence](#), [ID3D11Device5](#) and [ID3D11DeviceContext4](#) provide the same fence functionality as Direct3D 12 for Direct3D 11. Fences are used to synchronize multiple Direct3D11 devices, and for interop between Direct3D 11 and Direct3D 12. Fences are supported in the Windows 10 Creators Update.

Extended NV12 texture support

NV12 textures with capture and video encode capabilities now support sharing. Older D3D11 texture flags for video encode and capture are deprecated for NV12, as it will be set all the time for new drivers. Such textures can be shared not only with D3D11, but also with D3D12. In D3D12, no new flags represent these texture capabilities.

Refer to the boolean setting in [D3D11_FEATURE_DATA_D3D11_OPTIONS4](#).

Shader Caching

Drivers may support OS-managed shader caching of Direct3D11 applications in the Windows 10 Creators update.

Related topics

[What's new in Direct3D 11](#)

Features Introduced In Previous Releases

2/4/2021 • 2 minutes to read • [Edit Online](#)

Discover what new features have been added to the previous SDK updates:

In this section

TOPIC	DESCRIPTION
What's New in the August 2009 Windows 7/Direct3D 11 SDK	This version of Windows 7/Direct3D 11 ships as part of the DirectX SDK and contains new features, tools, and documentation.
What's New in the November 2008 Direct3D 11 Technical Preview	This version of Direct3D 11 contains new features, tools, and documentation.

Related topics

[What's new in Direct3D 11](#)

[Direct3D 11 Graphics](#)

What's New in the August 2009 Windows 7/Direct3D 11 SDK

11/2/2020 • 2 minutes to read • [Edit Online](#)

This version of Windows 7/Direct3D 11 ships as part of the DirectX SDK and contains new features, tools, and documentation.

ITEM	DESCRIPTION
Direct2D	<p>Direct2D is a hardware-accelerated, immediate-mode, 2-D graphics API that provides high performance and high quality rendering for 2-D geometry, bitmaps, and text. The Direct2D API is designed to interoperate well with Direct3D and GDI. This SDK allows developers to evaluate the API and write simple applications, with some of the more advanced functionality possible on properly configured machines. Documentation and samples for Direct2D are currently available on MSDN.</p>
DirectWrite	<p>DirectWrite provides support for high-quality text rendering, resolution-independent outline fonts, full Unicode text and layout support, and much, much more:</p> <ul style="list-style-type: none">• A device-independent text layout system that improves text readability in documents and in UI.• High-quality, sub-pixel, ClearType text rendering that can use GDI Direct3D, Direct2D, or application-specific rendering technology.• Support for multi-format text.• Support for the advanced typography features of OpenType fonts.• Support for the layout and rendering of text in all languages supported by Windows. <p>This SDK allows developers to evaluate the API and write basic applications for demonstration purposes only. Documentation and samples for DirectWrite are currently available on MSDN.</p>

ITEM	DESCRIPTION
DXGI 1.1	<p>DXGI 1.1 builds on DXGI 1.0 and will be available on both Windows Vista and Windows 7. DXGI 1.1 adds several new features:</p> <ul style="list-style-type: none"> • Synchronized Shared Surfaces Support. This enables efficient read and write surface sharing between multiple D3D (could be between D3D10 and D3D11) devices. • BGRA format support. This allows GDI to render to the same DXGI surface targeted by a Direct2D, Direct3D 10.1 or Direct3D 11 device. • Maximum Frame Latency. Using IDXGIDevice1::SetMaximumFrameLatency and IDXGIDevice1::GetMaximumFrameLatency, titles can control the number of frames that are allowed to be stored in a queue, before submission for rendering. Latency is often used to control how the CPU chooses between responding to user input and frames that are in the render queue. • Adapter Enumeration. Using IDXGIFactory1::EnumAdapters1, titles can enumerate local adapters without any monitors or outputs attached, as well as adapters with outputs attached.
Updated Samples	<p>This release has several new and updated samples.</p> <ul style="list-style-type: none"> • The new AdaptiveTessellationCS40 is an illustration of more advanced compute shader processing techniques that can be run on a D3D10 or D3D11 GPU. • The HDRToneMappingCS11 sample has been expanded to implement blur and bloom effects (in addition to tone mapping) using compute shader, as well as providing pixel shader implementations for comparison. • The MultithreadedRendering11 sample has been significantly updated, with more complex art assets and more intensive per-thread processing. • The SubD11 sample has been updated with a new facial model, and the sample now leverages the adjacency computation feature of the Samples Content Exporter.

Related topics

[Features Introduced In Previous Releases](#)

What's new in the Nov 2008 Direct3D 11 Tech Preview

11/2/2020 • 6 minutes to read • [Edit Online](#)

This version of Direct3D 11 contains new features, tools, and documentation.

ITEM	DESCRIPTION
Tessellation	Direct3D 11 provides additional pipeline stages to support real-time tessellation of high order primitives. With extensively programmable capabilities, this feature allows many different methods for evaluating high-order surfaces, including subdivision surfaces using approximation techniques, Bezier patches, adaptive tessellation, and displacement mapping. This feature will only be available on Direct3D 11-class hardware, so in order to evaluate this feature you will need to use the Reference Rasterizer. For a demo of tessellation in action, check out the SubD11 sample available through the Sample Browser.
Compute Shader	The Compute Shader is an additional stage independent of the Direct3D 11 pipeline that enables general purpose computing on the GPU. In addition to all shader features provided by the unified shader core, the Compute Shader also supports scattered reads and writes to resources through Unordered Access Views, a shared memory pool within a group of executing threads, synchronization primitives, atomic operators, and many other advanced data-parallel features. A variant of the Direct3D 11 Compute Shader has been enabled in this release that can operate on Direct3D 10-class hardware. It is therefore possible to develop Compute Shaders on actual hardware, but an updated driver is required. The full functionality of the Direct3D 11 Compute Shader is intended for support of Direct3D 11-class hardware, so in order to evaluate the full functionality you will need to use the Reference Rasterizer until such hardware is available. For a demo of the Compute Shader in action, check out the HDRToneMappingCS11 sample available through the Sample Browser.
Multithreaded Rendering	The key API difference from Direct3D 10 in Direct3D 11 is the addition of deferred contexts , which enables scalable execution of Direct3D commands distributed over multiple cores. A Deferred Context captures and assembles actions like state changes and draw submissions that can be executed on the actual device at a later time. By utilizing Deferred Contexts on multiple threads, an application can distribute the CPU overhead needed in the Direct3D11 runtime and the driver to multiple cores, enabling better use of an end-user's machine configuration. This feature is available for use on current Direct3D 10 hardware as well as the reference rasterizer. For a demonstration of API usage, check out the MultithreadedRendering11 sample available through the Sample Browser.

ITEM	DESCRIPTION
Dynamic Shader Linkage	<p>In order to address the combinatorial explosion problem seen in specializing shaders for performance, Direct3D 11 introduces a limited form of runtime shader linkage that allows for near-optimal shader specialization during execution of an application. This is achieved by specifying the implementations of specific functions in shader code when the shader is assigned to the pipeline, allowing the driver to inline native shader instructions quickly rather than forcing the driver to recompile the intermediate language into native instructions with the new configuration. Shader development is exposed through the introduction of classes and interfaces to HLSL. For a demonstration, check out the Dynamic Shader Linkage 11 sample available through the Sample Browser.</p>
Windows Advanced Rasterizer (WARP)	<p>Available in this SDK through Direct3D 11 and eventually also through Direct3D 10.1, WARP is a fast, multi-core scaling rasterizer that is fully Direct3D 10.1 compliant. Utilizing this technology is as simple as passing the D3D_DRIVER_TYPE_WARP flag in your device creation.</p>
Direct3D 10 and Direct3D 11 on Direct3D 9 Hardware (D3D10 Level 9)	<p>Available in this SDK through Direct3D 11 and eventually also through Direct3D 10.1, the Direct3D API can target most Direct3D 9 hardware as well as Direct3D 10, Direct3D 10.1 and Direct3D 11 hardware. This is achieved by providing the Feature Level mechanism, which groups hardware into six categories depending on functionality: 9_1, 9_2, 9_3, 10_0, 10_1 and 11_0. A card only meets a feature level if it is fully compliant to that level, and each level is a strict super-set of those below it. Functionality is minimally emulated to assure no unexpected performance cliffs are encountered. Thus, features like Geometry Shaders are not available for Direct3D 9 level targets.</p>
Runtime Binaries	<p>All runtime binaries provided in the Direct3D 11 tech preview that will be available on Windows 7 and Windows Vista SP1 are installed with the SDK and are labeled as "Beta" components (i.e. D3D11_beta.DLL). All beta-labeled components are time-bombed. To create projects to evaluate these new components, you must link to their equivalent beta-labeled import libraries (i.e. D3D11_beta.lib). If you have a PDC copy of Windows 7, the headers, libs, and pdbs provided in the Windows SDK with the build are appropriate for development using the Direct3D 11 components providing in Windows 7. Please reserve the use of the headers, libs, and pdbs in this SDK to the beta components provided herein.</p>
D3DX11	<p>D3DX11 currently supports texture loading, shader compilation, data loading and worker thread functions for Direct3D 11 resources. In the future, this component will provide more of the technologies available in D3DX10. Shader compilation functionality is also provided directly through the Direct3D Compiler component, described next.</p>

ITEM	DESCRIPTION
HLSL and Direct3D Compiler	<p>The HLSL compiler has several new features for supporting the new technologies available in Direct3D 11. This includes object oriented programming through interfaces and classes, a direct indexing syntax for resource loads, and the 'precise' keyword for ensuring that all operations performed with a specific variable adhere to the strict floating point rules.</p> <p>Almost all new linguistic features have valid functionality on existing shader targets. In addition to supporting all Direct3D 9, Direct3D 10, Direct3D 10.1, and Direct3D 11 shaders the HLSL compiler also supports the special targets needed to write shaders for Direct3D 10 Level 9 targets. The D3D Compiler is now directly accessible outside of D3DX10 and D3DX11 through D3DCompiler.h and D3DCompiler.lib. With these new files, an application is not required to link to D3DX in order to perform runtime compilation, and an application is not required to include the compiler if only D3DX functionality is needed.</p>
D3D11 Reference Rasterizer	<p>The Reference Rasterizer provides a gold-standard rasterization implementation for evaluation of Direct3D 11 features not yet available in hardware. The Reference Rasterizer is also provided as a way to verify a specific hardware implementation's accuracy to the rasterization standard. The reference rasterizer is designed for correctness, not performance. To create a reference device, simply pass the D3D_DRIVER_TYPE_REFERENCE flag at device creation.</p>
D3D11 SDK Layers	<p>Direct3D11 SDK Layers provide a mechanism for tracking the operation of the Direct3D 11 runtime during development. Currently this is used for providing useful debug information, which not only includes errors on improper use but also warnings that recommend best practice use of the runtime and often provides in-depth, useful information for debugging. It is highly recommended that the debug output from D3D11 SDK Layers is turned on at all times during development and an application generates no debug output during execution before it is released or used with PIX for Windows for profiling. Enabling the debug layer is as simple as passing the D3D11_CREATE_DEVICE_DEBUG flag at device creation time. Developers are strongly encouraged to use layers in debug builds. Layers are not recommended for use in profile or release builds.</p>

ITEM	DESCRIPTION
New Samples	<p>This release has four new samples.</p> <ul style="list-style-type: none">• The Dynamic Shader Linkage 11 sample demonstrates use of Shader Model 5 shader interfaces and Direct3D 11 support for linking shader interface methods at runtime.• The HDRToneMappingCS11 sample demonstrates how to setup and run the Compute Shader(CS for short later on), which is one of the most exciting new features of Direct3D 11. Although the sample only utilizes the CS to implement HDR tone-mapping, the concept should extend easily to other post-processing algorithms as well as more general calculations.• The MultithreadedRendering11 sample demonstrates how to split rendering among multiple threads, with very low overhead.• The new SubD11 sample is very similar to the SubD10 sample in the DirectX SDK, except that it has been enhanced to take advantage of three new Direct3D 11 pipeline stages: the hull shader, the tessellator, and the domain shader.

Related topics

[Features Introduced In Previous Releases](#)

Programming guide for Direct3D 11

11/2/2020 • 2 minutes to read • [Edit Online](#)

The programming guide contains information about how to use the Microsoft Direct3D 11 programmable pipeline to create realtime 3D graphics for games as well as scientific and desktop applications.

In this section

TOPIC	DESCRIPTION
Devices	This section describes Direct3D 11 device and device-context objects.
Resources	This section describes aspects of Direct3D 11 resources.
Graphics pipeline	This section describes the Direct3D 11 programmable pipeline.
Compute shader overview	A compute shader is a programmable shader stage that expands Direct3D 11 beyond graphics programming. The compute shader technology is also known as the DirectCompute technology.
Rendering	This section contains information about several Direct3D 11 rendering technologies.
Effects	A DirectX effect is a collection of pipeline state, set by expressions written in HLSL and some syntax that is specific to the effect framework.
Migrating to Direct3D 11	This section provides info for migrating to Direct3D 11 from an earlier version of Direct3D.
Direct3D video interfaces	This topic lists the Direct3D video interfaces that are available in Direct3D 9Ex and that are supported on Windows 7 and later versions of Windows client operating systems and on Windows Server 2008 R2 and later versions of Windows server operating systems.

Related topics

- [Direct3D 11 Graphics](#)
- [Graphics Reference](#)

Devices (Direct3D 11 Graphics)

2/4/2021 • 2 minutes to read • [Edit Online](#)

A Direct3D device allocates and destroys objects, renders primitives, and communicates with a graphics driver and the hardware. In Direct3D 11, a device is separated into a device object for creating resources and a device-context object, which performs rendering. This section describes Direct3D 11 device and device-context objects.

Objects created from one device cannot be used directly with other devices. Use a shared resource to share data between multiple devices, with the constraint that a shared object can be used only by the device that created it.

In this section

TOPIC	DESCRIPTION
Introduction to a Device in Direct3D 11	The Direct3D 11 object model separates resource creation and rendering functionality into a device and one or more contexts; this separation is designed to facilitate multithreading.
Software Layers	The Direct3D 11 runtime is constructed with layers, starting with the basic functionality at the core and building optional and developer-assist functionality in outer layers. This section describes the functionality of each layer.
Limitations Creating WARP and Reference Devices	Some limitations exist for creating WARP and Reference devices in Direct3D 10.1 and Direct3D 11.0. This topic discusses those limitations.
Direct3D 11 on Downlevel Hardware	This section discusses how Direct3D 11 is designed to support both new and existing hardware, from DirectX 9 to DirectX 11.
Using Direct3D 11 feature data to supplement Direct3D feature levels	Find out how to check device support for optional features, including features that were added in recent versions of Windows.

How to topics about devices

TOPIC	DESCRIPTION
How To: Create a Reference Device	Describes how to create a reference device.
How To: Create a WARP Device	Describes how to create a WARP device.
How To: Create a Swap Chain	Describes how to create a swap chain.
How To: Enumerate Adapters	Describes how to enumerate the physical display adapters.
How To: Get Adapter Display Modes	Describes how to get the supported display capabilities of an adapter.

TOPIC	DESCRIPTION
How To: Create a Device and Immediate Context	Describes how to initialize a device.

Related topics

[Programming Guide for Direct3D 11](#)

Introduction to a Device in Direct3D 11

2/22/2020 • 2 minutes to read • [Edit Online](#)

The Direct3D 11 object model separates resource creation and rendering functionality into a device and one or more contexts; this separation is designed to facilitate multithreading.

- [Device](#)
- [Device Context](#)
 - [Immediate Context](#)
 - [Deferred Context](#)
- [Threading Considerations](#)
- [Related topics](#)

Device

A device is used to create resources and to enumerate the capabilities of a display adapter. In Direct3D 11, a device is represented with an [ID3D11Device](#) interface.

Each application must have at least one device, most applications only create one device. Create a device for one of the hardware drivers installed on your machine by calling [D3D11CreateDevice](#) or [D3D11CreateDeviceAndSwapChain](#) and specify the driver type with the [D3D_DRIVER_TYPE](#) flag. Each device can use one or more device contexts, depending on the functionality desired.

Device Context

A device context contains the circumstance or setting in which a device is used. More specifically, a device context is used to set pipeline state and generate rendering commands using the resources owned by a device. Direct3D 11 implements two types of device contexts, one for immediate rendering and the other for deferred rendering; both contexts are represented with an [ID3D11DeviceContext](#) interface.

Immediate Context

An immediate context renders directly to the driver. Each device has one and only one immediate context which can retrieve data from the GPU. An immediate context can be used to immediately render (or play back) a [command list](#).

There are two ways to get an immediate context:

- By calling either [D3D11CreateDevice](#) or [D3D11CreateDeviceAndSwapChain](#).
- By calling [ID3D11Device::GetImmediateContext](#).

Deferred Context

A deferred context records GPU commands into a [command list](#). A deferred context is primarily used for multithreading and is not necessary for a single-threaded application. A deferred context is generally used by a worker thread instead of the main rendering thread. When you create a deferred context, it does not inherit any state from the immediate context.

To get a deferred context, call [ID3D11Device::CreateDeferredContext](#).

Any context -- immediate or deferred -- can be used on any thread as long as the context is only used in one thread at a time.

Threading Considerations

This table highlights the differences in the threading model in Direct3D 11 from prior versions of Direct3D.

Differences between Direct3D 11 and previous versions of Direct3D:

All [ID3D11Device](#) interface methods are free-threaded, which means it is safe to have multiple threads call the functions at the same time.

- All [ID3D11DeviceChild](#)-derived interfaces ([ID3D11Buffer](#), [ID3D11Query](#), etc.) are free-threaded.
- Direct3D 11 splits resource creating and rendering into two interfaces. Map, Unmap, Begin, End, and GetData are implemented on [ID3D11DeviceContext](#) because [ID3D11Device](#) strongly defines the order of operations. [ID3D11Resource](#) and [ID3D11Asynchronous](#) interfaces also implement methods for free-threaded operations.
- The [ID3D11DeviceContext](#) methods (except for those that exist on [ID3D11DeviceChild](#)) are not free-threaded, that is, they require single threading. Only one thread may safely be calling any of its methods (Draw, Copy, Map, etc.) at a time.
- In general, free-threading minimizes the number of synchronization primitives used as well as their duration. However, an application that uses synchronization held for a long time can directly impact how much concurrency an application can expect to achieve.

[ID3D10Device](#) interface methods are not designed to be free-threaded. [ID3D10Device](#) implements all create and rendering functionality (as does [ID3D9Device](#) in Direct3D 9). Map and Unmap are implemented on [ID3D10Resource](#)-derived interfaces, Begin, End, and GetData are implemented on [ID3D10Asynchronous](#)-derived interfaces.

Related topics

[Devices](#)

Software Layers

11/2/2020 • 2 minutes to read • [Edit Online](#)

The Direct3D 11 runtime is constructed with layers, starting with the basic functionality at the core and building optional and developer-assist functionality in outer layers. This section describes the functionality of each layer.

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.

To create layers when a device is created, call [D3D11CreateDevice](#) or [D3D11CreateDeviceAndSwapChain](#) and supply one or more [D3D11_CREATE_DEVICE_FLAG](#) values.

Direct3D 11 provides the following runtime layers:

- [Core Layer](#)
- [Debug Layer](#)
- [Related topics](#)

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.

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).

To create a device that supports the debug layer, you must install the DirectX SDK (to get D3D11SDKLayers.dll), and then specify the [D3D11_CREATE_DEVICE_DEBUG](#) flag when calling the [D3D11CreateDevice](#) function or the [D3D11CreateDeviceAndSwapChain](#) function. If you run your application with the debug layer enabled, the application will be substantially slower. But, to ensure that your application is clean of errors and warnings before you ship it, use the debug layer. For more info, see [Using the debug layer to debug apps](#).

NOTE

For Windows 7 with Platform Update for Windows 7 (KB2670838) or Windows 8.x, to create a device that supports the debug layer, install the Windows Software Development Kit (SDK) for Windows 8.x to get D3D11_1SDKLayers.dll.

NOTE

For Windows 10, to create a device that supports the debug layer, enable the "Graphics Tools" optional feature. Go to the Settings panel, under System, Apps & features, Manage optional Features, Add a feature, and then look for "Graphics Tools".

NOTE

For info about how to debug DirectX apps remotely, see [Debugging DirectX apps remotely](#).

Alternately, you can enable/disable the debug flag using the [DirectX Control Panel](#) included as part of the DirectX SDK.

When the debug layer lists memory leaks, it outputs a list of object interface pointers along with their friendly names. The default friendly name is "<unnamed>". You can set the friendly name by using the [ID3D11DeviceChild::SetPrivateData](#) method and the [WKPDID_D3DDescribeObjectName](#) GUID that is in D3DCommon.h. For example, to name pTexture with a SDKLayer name of mytexture.jpg, use the following code:

```
const char c_szName[] = "mytexture.jpg";
pTexture->SetPrivateData( WKPDID_D3DDescribeObjectName, sizeof( c_szName ) - 1, c_szName );
```

Typically, you should compile these calls out of your production version.

Related topics

[Devices](#)

Using the debug layer to debug apps

2/22/2020 • 3 minutes to read • [Edit Online](#)

We recommend that you use the [debug layer](#) to debug your apps to ensure that they are clean of errors and warnings. The debug layer helps you write Direct3D code. In addition, your productivity can increase when you use the debug layer because you can immediately see the causes of obscure rendering errors or even black screens at their source. The debug layer provides warnings for many issues. For example, the debug layer provides warnings for these issues:

- Forgot to set a texture but read from it in your pixel shader
- Output depth but have no depth-stencil state bound
- Texture creation failed with INVALIDARG

Here we talk about how to enable the [debug layer](#) and some of the issues that you can prevent by using the debug layer.

- [Enabling the debug layer](#)
- [Preventing errors in your app with the debug layer](#)
 - [Don't pass NULL pointers to Map](#)
 - [Confine source box within source and destination resources](#)
 - [Don't drop DiscardResource or DiscardView](#)
- [Related topics](#)

Enabling the debug layer

To enable the [debug layer](#), specify the `D3D11_CREATE_DEVICE_DEBUG` flag in the *Flags* parameter when you call the `D3D11CreateDevice` function to create the rendering device. This example code shows how to enable the debug layer when your Microsoft Visual Studio project is in a debug build:

```

    UINT creationFlags = D3D11_CREATE_DEVICE_BGRA_SUPPORT;
#if defined(_DEBUG)
    // If the project is in a debug build, enable the debug layer.
    creationFlags |= D3D11_CREATE_DEVICE_DEBUG;
#endif
    // Define the ordering of feature levels that Direct3D attempts to create.
    D3D_FEATURE_LEVEL featureLevels[] =
    {
        D3D_FEATURE_LEVEL_11_1,
        D3D_FEATURE_LEVEL_11_0,
        D3D_FEATURE_LEVEL_10_1,
        D3D_FEATURE_LEVEL_10_0,
        D3D_FEATURE_LEVEL_9_3,
        D3D_FEATURE_LEVEL_9_1
    };

    ComPtr<ID3D11Device> d3dDevice;
    ComPtr<ID3D11DeviceContext> d3dDeviceContext;
    DX::ThrowIfFailed(
        D3D11CreateDevice(
            nullptr, // specify nullptr to use the default adapter
            D3D_DRIVER_TYPE_HARDWARE,
            nullptr, // specify nullptr because D3D_DRIVER_TYPE_HARDWARE
            // indicates that this function uses hardware
            creationFlags, // optionally set debug and Direct2D compatibility flags
            featureLevels,
            ARRAYSIZE(featureLevels),
            D3D11_SDK_VERSION, // always set this to D3D11_SDK_VERSION
            &d3dDevice,
            nullptr,
            &d3dDeviceContext
        )
    );

```

Preventing errors in your app with the debug layer

If you misuse the Direct3D 11 API or pass bad parameters, the debug output of the [debug layer](#) reports an error or a warning. You can then correct your mistake. Next, we look at some coding issues that can cause undefined behavior or even the operating system to crash. You can catch and prevent these issues by using the debug layer.

Don't pass NULL pointers to Map

If you pass **NULL** to the *pResource* or *pMappedResource* parameter of the [ID3D11DeviceContext::Map](#) method, the behavior of **Map** is undefined. If you created a device that just supports the [core layer](#), invalid parameters to **Map** can crash the operating system. If you created a device that supports the [debug layer](#), the debug output reports an error on this invalid **Map** call.

Confine source box within source and destination resources

In a call to the [ID3D11DeviceContext::CopySubresourceRegion](#) method, the source box must be within the source resource. The destination offsets, (x, y, and z) allow the source box to be offset when writing into the destination resource, but the dimensions of the source box and the offsets must be within the size of the resource. If you try to copy outside the destination resource or specify a source box that is larger than the source resource, the behavior of **CopySubresourceRegion** is undefined. If you created a device that supports the [debug layer](#), the debug output reports an error on this invalid **CopySubresourceRegion** call. Invalid parameters to **CopySubresourceRegion** cause undefined behavior and might result in incorrect rendering, clipping, no copy, or even the removal of the rendering device.

Don't drop DiscardResource or DiscardView

The runtime drops a call to [ID3D11DeviceContext1::DiscardResource](#) or

[ID3D11DeviceContext1::DiscardView](#) unless you correctly create the resource.

The resource that you pass to [ID3D11DeviceContext1::DiscardResource](#) must have been created by using [D3D11_USAGE_DEFAULT](#) or [D3D11_USAGE_DYNAMIC](#), otherwise the runtime drops the call to [DiscardResource](#).

The resource that underlies the view that you pass to [ID3D11DeviceContext1::DiscardView](#) must have been created using [D3D11_USAGE_DEFAULT](#) or [D3D11_USAGE_DYNAMIC](#), otherwise the runtime drops the call to [DiscardView](#).

If you created a device that supports the [debug layer](#), the debug output reports an error regarding the dropped call.

Related topics

[Software Layers](#)

Limitations Creating WARP and Reference Devices

11/2/2020 • 2 minutes to read • [Edit Online](#)

Some limitations exist for creating WARP and Reference devices in Direct3D 10.1 and Direct3D 11.0. This topic discusses those limitations.

The D3D10_DRIVER_TYPE_WARP and D3D10_DRIVER_TYPE_REFERENCE driver types are not supported on the D3D10_FEATURE_LEVEL_9_1 through D3D10_FEATURE_LEVEL_9_3 feature levels in Direct3D 10.1. Additionally, the D3D_DRIVER_TYPE_WARP driver type is not supported on D3D_FEATURE_LEVEL_11_0 in Direct3D 11.0. That is, when you call [D3D10CreateDevice1](#) to create a Direct3D 10.1 device or when you call [D3D11CreateDevice](#) to create a Direct3D 11.0 device, if you specify a combination of one of these driver types with one of these feature levels in the call, the call is invalid. Only the following combinations of feature levels, runtimes, and driver types are valid for WARP and Reference devices:

- D3D_DRIVER_TYPE_WARP on all feature levels in Direct3D 11.1, which Windows 8 includes D3D_DRIVER_TYPE_REFERENCE on all feature levels in Direct3D 11.1

When you call [D3D11CreateDevice](#) to create a Direct3D 11.1 device, the call is valid if you specify a combination of one of these driver types with one of these feature levels.

- D3D_DRIVER_TYPE_WARP on D3D_FEATURE_LEVEL_9_1 through D3D_FEATURE_LEVEL_10_1 feature levels in Direct3D 11
- D3D_DRIVER_TYPE_REFERENCE on D3D_FEATURE_LEVEL_9_1 through D3D_FEATURE_LEVEL_11_0 feature levels in Direct3D 11

When you call [D3D11CreateDevice](#) to create a Direct3D 11 device, the call is valid if you specify a combination of one of these driver types with one of these feature levels.

- D3D10_DRIVER_TYPE_WARP and D3D10_DRIVER_TYPE_REFERENCE on D3D10_FEATURE_LEVEL_10_0 through D3D10_FEATURE_LEVEL_10_1 feature levels in Direct3D 10.1

When you call [D3D10CreateDevice1](#) to create a Direct3D 10.1 device, the call is valid if you specify a combination of one of these driver types with one of these feature levels.

Related topics

[Devices](#)

[Introduction to Direct3D 11 on Downlevel Hardware](#)

[How To: Create a WARP Device](#)

[How To: Create a Reference Device](#)

[D3D10_DRIVER_TYPE](#)

[D3D10_FEATURE_LEVEL1](#)

[D3D_DRIVER_TYPE](#)

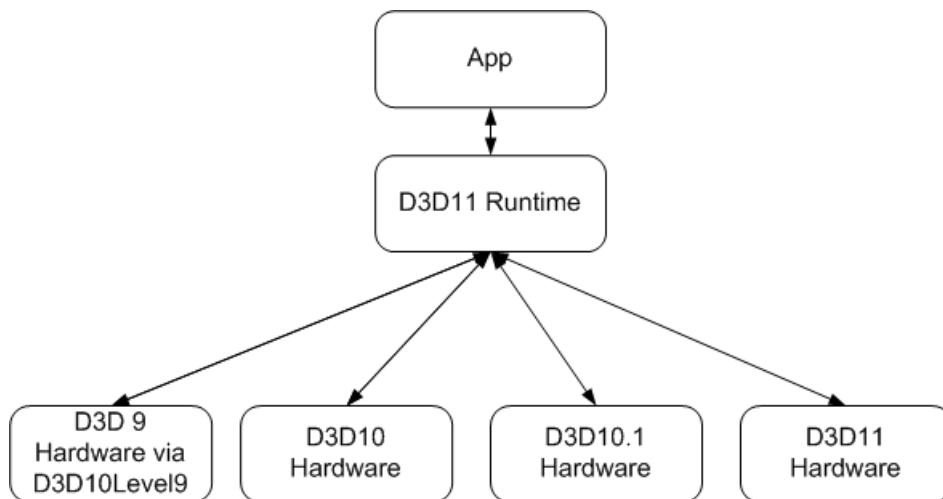
[D3D_FEATURE_LEVEL](#)

Direct3D 11 on Downlevel Hardware

2/4/2021 • 2 minutes to read • [Edit Online](#)

This section discusses how Direct3D 11 is designed to support both new and existing hardware, from DirectX 9 to DirectX 11.

This diagram shows how Direct3D 11 supports new and existing hardware.



With Direct3D 11, a new paradigm is introduced called feature levels. A feature level is a well defined set of GPU functionality. Using a feature level, you can target a Direct3D application to run on a downlevel version of Direct3D hardware.

The [10Level9 Reference](#) section lists the differences between how various [ID3D11Device](#) and [ID3D11DeviceContext](#) methods behave at various 10Level9 feature levels.

In this section

TOPIC	DESCRIPTION
Direct3D feature levels	This topic discusses Direct3D feature levels.
Exceptions	This topic describes exceptions when using Direct3D 11 on downlevel hardware.
Compute Shaders on Downlevel Hardware	This topic discusses how to make use of compute shaders in a Direct3D 11 app on Direct3D 10 hardware.
Preventing Unwanted NULL Pixel Shader SRVs	This topic discusses how to work around the driver receiving NULL shader-resource views (SRVs) even when non-NULL SRVs are bound to the pixel shader stage.

How to topics about feature levels

TOPIC	DESCRIPTION
How To: Get the Device Feature Level	How to get a feature level.

Related topics

[Devices](#)

Direct3D feature levels

11/2/2020 • 8 minutes to read • [Edit Online](#)

To handle the diversity of video cards in new and existing machines, Microsoft Direct3D 11 introduces the concept of feature levels. This topic discusses Direct3D feature levels.

Each video card implements a certain level of Microsoft DirectX (DX) functionality depending on the graphics processing units (GPUs) installed. In prior versions of Microsoft Direct3D, you could find out the version of Direct3D the video card implemented, and then program your application accordingly.

With Direct3D 11, a new paradigm is introduced called feature levels. A feature level is a well-defined set of GPU functionality. For instance, the 9_1 feature level implements the functionality that was implemented in Microsoft Direct3D 9, which exposes the capabilities of shader models `ps_2_x` and `vs_2_x`, while the 11_0 feature level implements the functionality that was implemented in Direct3D 11.

Now when you create a device, you can attempt to create a device for the feature level that you want to request. If the device creation works, that feature level exists, if not, the hardware does not support that feature level. You can either try to recreate a device at a lower feature level or you can choose to exit the application. For more info about creating a device, see the [D3D11CreateDevice](#) function.

Using feature levels, you can develop an application for Direct3D 9, Microsoft Direct3D 10, or Direct3D 11, and then run it on 9, 10 or 11 hardware (with some exceptions; for example, new 11 features will not run on an existing 9 card). Here is a couple of other basic properties of feature levels:

- A GPU that allows a device to be created meets or exceeds the functionality of that feature level.
- A feature level always includes the functionality of previous or lower feature levels.
- A feature level does not imply performance, only functionality. Performance is dependent on hardware implementation.
- Choose a feature level when you create a Direct3D 11 device.

For information about limitations creating nonhardware-type devices on certain feature levels, see [Limitations Creating WARP and Reference Devices](#).

To assist you in deciding what feature level to design with, compare the features for each feature level.

The [10Level9 Reference](#) section lists the differences between how various [ID3D11Device](#) and [ID3D11DeviceContext](#) methods behave at various 10Level9 feature levels.

Formats of version numbers

There are three formats for Direct3D versions, shader models, and feature levels.

- Direct3D versions use a period; for example, Direct3D 12.0.
- Shader models use a period; for example, shader model 5.1.
- Feature levels use an underscore; for example, feature level 12_0.

Feature support for feature levels 12_1 through 9_3

The following features are available for the feature levels listed. The headings across the top row are Direct3D feature levels. The headings in the left-hand column are features. Also see [Footnotes for the tables](#).

FEATURE \ FEATURE LEVEL	12_1 ⁰	12_0 ⁰	11_1 ¹	11_0	10_1	10_0	9_3 ⁷
Shader Model (D3D11)	5.0 ²	5.0 ²	5.0 ²	5.0 ²	4.x	4.0	2.0 (4_0_level_9_3) [vs_2_a/ps_2_x] ⁵
Shader Model (D3D12)	5.1 ²	5.1 ²	5.1 ²	5.1 ²	N/A	N/A	N/A
Tiled resources	Tier2 ⁶	Tier2 ⁶	Optional	Optional	No	No	No
Conservative Rasterization	Tier1 ⁶	Optional	Optional	No	No	No	No
Rasterizer Order Views	Yes	Optional	Optional	No	No	No	No
Min/Max Filters	Yes	Yes	Optional	No	No	No	No
Map Default Buffer	Optional	Optional	Optional	Optional	No	No	No
Shader Specified Stencil Reference Value	Optional	Optional	Optional	No	No	No	No
Typed Unordered Access View Loads	18 formats, more optional	18 formats, more optional	3 formats, more optional	3 formats, more optional	No	No	No
Geometry Shader	Yes	Yes	Yes	Yes	Yes	Yes	No
Stream Out	Yes	Yes	Yes	Yes	Yes	Yes	No
DirectCompute / Compute Shader	Yes	Yes	Yes	Yes	Optional	Optional	N/A
Feature \ Feature Level	12_1 ⁰	12_0 ⁰	11_1 ¹	11_0	10_1	10_0	9_3 ⁷

FEATURE \ FEATURE LEVEL	12_1	12_0	11_1	11_0	10_1	10_0	9_3
Hull and Domain Shaders	Yes	Yes	Yes	Yes	No	No	No
Texture Resource Arrays	Yes	Yes	Yes	Yes	Yes	Yes	No
Cubemap Resource Arrays	Yes	Yes	Yes	Yes	Yes	No	No
BC4/BC5 Compression	Yes	Yes	Yes	Yes	Yes	Yes	No
BC6H/BC7 Compression	Yes	Yes	Yes	Yes	No	No	No
Alpha-to-coverage	Yes	Yes	Yes	Yes	Yes	Yes	No
Extended Formats (BGRA, and so on)	Yes	Yes	Yes	Yes	Optional	Optional	Yes
10-bit XR High Color Format	Yes	Yes	Yes	Yes	Optional	Optional	N/A
Logic Operations (Output Merger)	Yes	Yes	Yes	Optional ¹	Optional ¹	Optional ¹	No
Target-independent rasterization	Yes	Yes	Yes	No	No	No	No
Multiple render target(MRT) with ForcedSampleCount 1	Yes	Yes	Yes	Optional ¹	Optional ¹	Optional ¹	No
UAV slots	64	64	64	8	1	1	N/A

FEATURE \ FEATURE LEVEL	12_1	12_0	11_1	11_0	10_1	10_0	9_3
Feature \ Feature Level	12_1 ⁰	12_0 ⁰	11_1 ¹	11_0	10_1	10_0	9_3 ⁷
UAVs at every stage	Yes	Yes	Yes	No	No	No	N/A
Max forced sample count for UAV-only rendering	16	16	16	8	N/A	N/A	N/A
Constant buffer offsetting and partial updates	Yes	Yes	Yes	Optional ¹	Optional ¹	Optional ¹	Yes ¹
16 bits per pixel (bpp) formats	Yes	Yes	Yes	Optional ¹	Optional ¹	Optional ¹	Optional ¹
Max Texture Dimension	16384	16384	16384	16384	8192	8192	4096
Max Cubemap Dimension	16384	16384	16384	16384	8192	8192	4096
Max Volume Extent	2048	2048	2048	2048	2048	2048	256
Max Texture Repeat	16384	16384	16384	16384	8192	8192	8192
Max Anisotropy	16	16	16	16	16	16	16
Max Primitive Count	2 ³² – 1	2 ³² – 1	2 ³² – 1	1048575			
Max Vertex Index	2 ³² – 1	2 ³² – 1	2 ³² – 1	1048575			
Max Input Slots	32	32	32	32	32	16	16

FEATURE \ FEATURE LEVEL	12_1	12_0	11_1	11_0	10_1	10_0	9_3
Simultaneous Render Targets	8	8	8	8	8	8	4
Occlusion Queries	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Feature \ Feature Level	12_1 ⁰	12_0 ⁰	11_1 ¹	11_0	10_1	10_0	9_3 ⁷
Separate Alpha Blend	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Mirror Once	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Overlapping Vertex Elements	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Independent Write Masks	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Instancing	Yes	Yes	Yes	Yes	Yes	Yes	Yes ⁷
Nonpowers-of-2 conditionally ³	No	No	No	No	No	No	Yes
Nonpowers-of-2 unconditionally ⁴	Yes	Yes	Yes	Yes	Yes	Yes	No

Feature support for feature levels 9_2 and 9_1

The following features are available for the feature levels listed. The headings across the top row are Direct3D feature levels. The headings in the left-hand column are features. Also see [Footnotes for the tables](#).

FEATURE \ FEATURE LEVEL	9_2	9_1
Shader Model (D3D11)	2.0 (4_0_level_9_1)	2.0 (4_0_level_9_1)
Shader Model (D3D12)	N/A	N/A
Tiled resources	No	No
Conservative Rasterization	No	No

FEATURE \ FEATURE LEVEL	9_2	9_1
Rasterizer Order Views	No	No
Min/Max Filters	No	No
Map Default Buffer	No	No
Shader Specified Stencil Reference Value	No	No
Typed Unordered Access View Loads	No	No
Geometry Shader	No	No
Stream Out	No	No
DirectCompute / Compute Shader	N/A	N/A
Hull and Domain Shaders	No	No
Texture Resource Arrays	No	No
Cubemap Resource Arrays	No	No
BC4/BC5 Compression	No	No
Feature \ Feature Level	9_2	9_1
BC6H/BC7 Compression	No	No
Alpha-to-coverage	No	No
Extended Formats (BGRA, and so on)	Yes	Yes
10-bit XR High Color Format	N/A	N/A
Logic Operations (Output Merger)	No	No
Target-independent rasterization	No	No
Multiple render target(MRT) with ForcedSampleCount 1	No	No
UAV slots	N/A	N/A
UAVs at every stage	N/A	N/A
Max forced sample count for UAV-only rendering	N/A	N/A
Constant buffer offsetting and partial updates	Yes ¹	Yes ¹

FEATURE \ FEATURE LEVEL	9_2	9_1
16 bits per pixel (bpp) formats	Optional ¹	Optional ¹
Max Texture Dimension	2048	2048
Max Cubemap Dimension	512	512
Max Volume Extent	256	256
Max Texture Repeat	2048	128
Feature \ Feature Level	9_2	9_1
Max Anisotropy	16	2
Max Primitive Count	1048575	65535
Max Vertex Index	1048575	65534
Max Input Slots	16	16
Simultaneous Render Targets	1	1
Occlusion Queries	Yes	No
Separate Alpha Blend	Yes	No
Mirror Once	Yes	No
Overlapping Vertex Elements	Yes	No
Independent Write Masks	No	No
Instancing	No	No
Nonpowers-of-2 conditionally ³	Yes	Yes
Nonpowers-of-2 unconditionally ⁴	No	No

Footnotes for the tables

⁰ Requires the Direct3D 11.3 or Direct3D 12 runtime.

¹ Requires the Direct3D 11.1 runtime.

² Shader model 5.0 and above can optionally support double-precision shaders, extended double-precision shaders, the **SAD4** shader instruction, and partial-precision shaders. To determine the shader model 5.0 options that are available for DirectX 11, call [ID3D11Device::CheckFeatureSupport](#). Some compatibility depends on what hardware you are running on. Shader model 5.1 and above are only supported through the DirectX 12 API, regardless of the feature level that's being used. DirectX 11 only supports up to shader model 5.0. The DirectX 12 API only goes down to feature level 11_0.

³ At feature levels 9_1, 9_2 and 9_3, the display device supports the use of 2-D textures with dimensions that are not powers of two under two conditions. First, only one MIP-map level for each texture can be created, and second, no wrap sampler modes for textures are allowed (that is, the **AddressU**, **AddressV**, and **AddressW** members of **D3D11_SAMPLER_DESC** cannot be set to **D3D11_TEXTURE_ADDRESS_WRAP**).

⁴ At feature levels 10_0, 10_1 and 11_0, the display device unconditionally supports the use of 2-D textures with dimensions that are not powers of two.

⁵ Vertex Shader 2a with 256 instructions, 32 temporary registers, static flow control of depth 4, dynamic flow control of depth 24, and D3DVS20CAPS_PREDICATION. Pixel Shader 2x with 512 instructions, 32 temporary registers, static flow control of depth 4, dynamic flow control of depth 24, D3DPS20CAPS_ARBITRARYSWIZZLE, D3DPS20CAPS_GRADIENTINSTRUCTIONS, D3DPS20CAPS_PREDICATION, D3DPS20CAPS_NODEPENDENTREADLIMIT, and D3DPS20CAPS_NOTEWINSTRUCTIONLIMIT.

⁶ Higher tiers optional.

⁷ For Feature Level 9_3, the only rendering methods supported are **Draw**, **DrawIndexed**, and **DrawIndexInstanced**. Also for Feature Level 9_3, point list rendering is supported only for rendering via **Draw**.

For details of format support at different hardware feature levels, refer to:

- [DXGI Format Support for Direct3D Feature Level 12.1 Hardware](#)
- [DXGI Format Support for Direct3D Feature Level 12.0 Hardware](#)
- [DXGI Format Support for Direct3D Feature Level 11.1 Hardware](#)
- [DXGI Format Support for Direct3D Feature Level 11.0 Hardware](#)
- [Hardware Support for Direct3D 10Level9 Formats](#)
- [Hardware Support for Direct3D 10.1 Formats](#)
- [Hardware Support for Direct3D 10 Formats](#)

Related topics

- [Direct3D 11 on Downlevel Hardware](#)
- [Hardware Feature Levels \(Direct3D 12\)](#)

Exceptions

11/2/2020 • 2 minutes to read • [Edit Online](#)

Some features of Direct3D 11 are not fully specified by feature levels. This topic describes exceptions when using Direct3D 11 on downlevel hardware. Perhaps a feature was added after the feature level is defined (and requires an updated driver) or perhaps different GPUs implement widely different implementations. Feature level exceptions can be gathered into the following groups:

- [Extended Formats](#)
- [Multisample Anti-Aliasing](#)
- [Texture2D Sizes](#)
- [Special Behavior of Adapters for Feature Level 9](#)
- [Related topics](#)

The [10Level9 Reference](#) section lists the differences between how various [ID3D11Device](#) and [ID3D11DeviceContext](#) methods behave at various 10Level9 feature levels.

Extended Formats

An extended format is a pixel format added to Direct3D 10.1 and Direct3D 11 for feature levels 10_0 and 10_1.

An extended format requires an updated driver (for Direct3D 10_1 or below). Use

[ID3D11Device::CheckFormatSupport](#) and [ID3D11Device::CheckFeatureSupport](#) to query for support for these extended formats.

An extended format:

- Adds support for BGRA order of 8-bit per-component resources.
- Allows casting of an integer-value swap-chain buffer. This allows an application to add or remove the _SRGB suffix or render to an XR_BIAS swap chain.
- Adds optional support for DXGI_FORMAT_R10G10B10_XR_BIAS_A2_UNORM.
- Guarantees that a DXGI_FORMAT_R16G16B16A16_FLOAT swap chain is presented as if the data contained is not sRGB-encoded.

The full set of extended formats are either fully supported or not supported, with the exception of the XR_BIAS format. The XR_BIAS format is:

- Unsupported in any 9 level
- Optional in either the 10_0 or 10_1 level
- Guaranteed at the 11_0 level

Multisample Anti-Aliasing

MSAA implementations have little in common across GPU implementations. Feature Level 10.1 added some well-defined minima, but at lower feature levels, MSAA must be tested explicitly using [ID3D11Device::CheckMultisampleQualityLevels](#).

Texture2D Sizes

A feature level guarantees that a minimum size can be created, however, an application can create larger textures up to the full size supported by the GPU. An application should expect failure from a method such as

[ID3D11Device::CreateTexture2D](#) if a maximum is exceeded.

Special Behavior of Adapters for Feature Level 9

The three lowest feature levels D3D_FEATURE_LEVEL_9_1, D3D_FEATURE_LEVEL_9_2 and D3D_FEATURE_LEVEL_9_3, share a common implementation DLL and treat the [IDXGIAdapter](#) argument to `D3D11CreateDevice[AndSwapchain]` as a template adapter and create their own adapter as part of device creation. This means that the [IDXGIAdapter](#) passed into the creation routine will not be the same adapter as that retrieved from the device via `IDXGIDevice::GetAdapter`. The impact of this is that the [IDXGIOOutputs](#) enumerated from the passed-in adapter cannot be used to enter fullscreen using any level 9 device, since those outputs are not owned by the device's adapter. It is good practice to discard the passed-in template adapter and retrieve the device's created adapter using `IDXGIDevice::GetAdapter`, where [IDXGIDevice](#) can be retrieved using `QueryInterface` from the Direct3D device interface.

Related topics

[Direct3D 11 on Downlevel Hardware](#)

Compute Shaders on Downlevel Hardware

11/2/2020 • 3 minutes to read • [Edit Online](#)

Direct3D 11 provides the ability to use [compute shaders](#) that operate on most Direct3D 10.x hardware, with some limitations to operation. The compute shader technology is also known as the DirectCompute technology. This topic discusses how to make use of [compute shaders](#) in a Direct3D 11 app on Direct3D 10 hardware.

Support for compute shaders on downlevel hardware is only for devices compatible with Direct3D 10.x. Compute shaders cannot be used on Direct3D 9.x hardware.

To check if Direct3D 10.x hardware supports compute shaders, call [ID3D11Device::CheckFeatureSupport](#). In the `CheckFeatureSupport` call, pass the `D3D11_FEATURE_D3D10_X_HARDWARE_OPTIONS` value to the `Feature` parameter, pass a pointer to the [D3D11_FEATURE_DATA_D3D10_X_HARDWARE_OPTIONS](#) structure to the `pFeatureSupportData` parameter, and pass the size of the `D3D11_FEATURE_DATA_D3D10_X_HARDWARE_OPTIONS` structure to the `FeatureSupportDataSize` parameter. `CheckFeatureSupport` returns `TRUE` in the `ComputeShaders_Plus_RawAndStructuredBuffers_Via_Shader_4_x` member of `D3D11_FEATURE_DATA_D3D10_X_HARDWARE_OPTIONS` if the Direct3D 10.x hardware supports compute shaders.

The [10Level9 Reference](#) section lists the differences between how various [ID3D11Device](#) and [ID3D11DeviceContext](#) methods behave at various 10Level9 feature levels.

- [Unordered Access Views \(UAVs\)](#)
- [Shader Resource Views \(SRVs\)](#)
- [Thread Groups](#)
 - [Thread Group Dimensions](#)
 - [Two-Dimensional Thread Indices](#)
 - [Thread Group Shared Memory \(TGSM\)](#)
- [D3DCompile with D3DCOMPILE_SKIP_OPTIMIZATION](#)
- [Related topics](#)

Unordered Access Views (UAVs)

Raw ([RWByteAddressBuffer](#)) and Structured ([RWStructuredBuffer](#)) Unordered Access Views are supported on downlevel hardware, with the following limitations:

- Only a single UAV may be bound to a pipeline at a time through [ID3D11DeviceContext::CSSetUnorderedAccessViews](#).
- The base offset for a Raw UAV must be aligned on a 256-byte boundary (instead of 16-byte alignment required for Direct3D 11 hardware).

Typed UAVs are not supported on downlevel hardware. This includes [Texture1D](#), [Texture2D](#), and [Texture3D](#) UAVs.

Pixel Shaders on downlevel hardware do not support unordered access.

Shader Resource Views (SRVs)

Raw and Structured Buffers as Shader Resource Views are supported on downlevel hardware for read-only access, as they are on Direct3D 11 hardware. These resource types are supported for Vertex Shaders, Geometry Shaders, Pixel Shaders as well as Compute Shaders.

Thread Groups

A compute shader can execute on many threads in parallel, within a thread group.

Thread groups are supported on downlevel hardware, with the following limitations:

Thread Group Dimensions

Thread groups defined for downlevel hardware are limited to X and Y dimensions of 768. This is less than the maximum values of 1024 for Direct3D 11 hardware. The maximum Z dimension of 64 is unchanged.

The total number of threads in the group ($X \times Y \times Z$) is limited to 768. This is less than the limit of 1024 for Direct3D 11 hardware.

If these numbers are exceeded, shader compilation will fail.

Two-Dimensional Thread Indices

A particular thread within a thread group is indexed using a 3D vector given by (x,y,z).

For compute shaders operating on downlevel hardware, thread groups only support two dimensions. This means that the Z value in the 3D vector must always be 1.

This limitation specifically applies to the following:

- **ID3D11DeviceContext::Dispatch**— The *ThreadGroupCountZ* argument must be 1.
- **ID3D11DeviceContext::DispatchIndirect**— This function is not supported on downlevel hardware.
- **numthreads**— The Z value must be 1.

Thread Group Shared Memory (TGSM)

Thread Group Shared Memory is limited to 16Kb on downlevel hardware. This is less than the 32Kb that is available to Direct3D 11 hardware.

A Compute Shader thread may only write to its own region of TGSM. This write-only region has a maximum size of 256 bytes or less, with the maximum decreasing as the number of threads declared for the group increases.

The following table defines the per-thread maximum size of a TGSM region for the number of threads in the group:

NUMBER OF THREADS IN GROUP	MAXIMUM TGSM SIZE PER THREAD
0-64	256
65-68	240
69-72	224
73-76	208
77-84	192
85-92	176
93-100	160
101-112	144
113-128	128

NUMBER OF THREADS IN GROUP	MAXIMUM TGSM SIZE PER THREAD
129-144	112
145-168	96
169-204	80
205-256	64
257-340	48
341-512	32
513-768	16

A Compute Shader thread may read the TGSM from any location.

D3DCompile with D3DCOMPILE_SKIP_OPTIMIZATION

[D3DCompile](#) returns E_NOTIMPL when you pass cs_4_0 as the shader target along with the [D3DCOMPILE_SKIP_OPTIMIZATION](#) compile option. The cs_5_0 shader target works with [D3DCOMPILE_SKIP_OPTIMIZATION](#).

Related topics

[Direct3D 11 on Downlevel Hardware](#)

Preventing Unwanted NULL Pixel Shader SRVs

2/22/2020 • 2 minutes to read • [Edit Online](#)

Direct3D 11 applications that run on Direct3D 9 graphics hardware could inadvertently cause the driver to receive **NULL** shader-resource views (SRVs) even when the applications bind non-**NULL** SRVs to the pixel shader stage. This situation can occur only if the applications destroy SRVs while they execute. This topic discusses how to work around the driver receiving **NULL** shader-resource views (SRVs) even when non-**NULL** SRVs are bound to the pixel shader stage.

To prevent the driver from receiving unwanted **NULL** SRVs, the applications must call [**ID3D11DeviceContext::PSSetShaderResources**](#) to unset all SRVs before each call to [**ID3D11DeviceContext::PSSetShader**](#). However, if the applications do not destroy SRVs until the end of their code execution, they do not need to unset the SRVs.

The [10Level9 Reference](#) section lists the differences between how various [**ID3D11Device**](#) and [**ID3D11DeviceContext**](#) methods behave at various 10Level9 feature levels.

Related topics

[Direct3D 11 on Downlevel Hardware](#)

Using Direct3D 11 feature data to supplement Direct3D feature levels

11/2/2020 • 5 minutes to read • [Edit Online](#)

Find out how to check device support for optional features, including features that were added in recent versions of Windows.

Direct3D feature levels indicate well-defined sets of GPU functionality that roughly correspond to different generations of graphics hardware. This greatly simplifies the task of checking hardware capabilities, and also provides a consistent experience across a wide array of different devices.

To account for some of the variance across different hardware implementations - including legacy hardware, mobile hardware, and modern hardware - some features are considered optional. Support for these features can be determined by calling [ID3D11Device::CheckFeatureSupport](#) and supplying the relevant D3D11_FEATURE_DATA_* structure. This topic describes the various optional Direct3D 11 features, how some of them work together, and how you can avoid checking for every single optional feature.

How to check for optional feature support

Call [ID3D11Device::CheckFeatureSupport](#), providing the structure that represents the optional feature you'd like to use. If the method returns S_OK, that means you're on a version of the Direct3D runtime that supports the optional feature. If it returns E_INVALIDARG, that means you're on a version of the Direct3D 11 runtime from before the optional feature was added - this means the optional feature is not available, along with any other optional features introduced in the same version of Direct3D 11 or later.

Can I minimize the work required for feature support checks?

In addition to having the right Direct3D 11 runtime (usually associated with a Windows version) the graphics driver must also be recent enough to support the optional feature. The WDDM specifications require optional features to be supported if the hardware can support it. So when a graphics driver supports one of the optional features that were added in a particular version of Windows, it usually means that the graphics driver supports the other features added in that version of Windows. For example, if a device driver supports shadows on feature level 9, then you know the device driver is at least WDDM 1.2.

Note If a Microsoft Direct3D device supports [feature level](#) 11.1, all of the optional features indicated by [D3D11_FEATURE_DATA_D3D11_OPTIONS](#) are automatically supported except [SAD4ShaderInstructions](#) and [ExtendedDoublesShaderInstructions](#).

The runtime always sets the following groupings of members identically. That is, all the values in a grouping are TRUE or FALSE together:

- DiscardAPIsSeenByDriver and FlagsForUpdateAndCopySeenByDriver
- ClearView, CopyWithOverlap, ConstantBufferPartialUpdate, ConstantBufferOffsetting, and MapNoOverwriteOnDynamicConstantBuffer
- MapNoOverwriteOnDynamicBufferSRV and MultisampleRTVWithForcedSampleCountOne

Feature level 11.2 options ([D3D11_FEATURE_DATA_D3D11_OPTIONS1](#)): The optional features indicated by this field are independent and must be checked individually.

Feature support on Windows RT 8.1 and Windows Phone 8.1 devices

Windows RT tablet devices can support a variety of feature levels and optional features, are optimized for

reduced power consumption, and use integrated graphics instead of discrete GPUs. Windows Store apps for ARM devices must support feature level 9.1. DirectX apps for Windows RT should take advantage of optional features that can save power and cycles - such as simple instancing - when they are available.

Windows Phone 8 mobile devices support feature level 9.3 with specific optional features. See [Direct3D feature level 9_3 for Windows Phone 8](#).

What are the Direct3D 11 optional features?

The rest of this article describes the optional features available in Direct3D 11.2. Features are described in chronological order by when they were added so you can get a sense for what features are in different versions of Direct3D 11.

Optional compute shader support for feature level 10

The following feature is always available for feature level 10 devices:

D3D11_FEATURE_DATA_D3D10_X_HARDWARE_OPTIONS: If this is TRUE, the device supports compute shaders. This includes support for raw and structured buffers.

When the feature level 10_0 or 10_1 device supports this feature, the device is not guaranteed to support compute shader 4.1. Apps should be prepared to fall back to a compute shader 4.0 if [ID3D11Device::CreateComputeShader](#) throws an exception with a compute shader 4.1 program.

Optional capabilities for feature level 9

The following features are added for feature level 9 starting in Windows 8:

D3D11_FEATURE_DATA_D3D9_OPTIONS: Indicates support for wrap texture addressing with non-power-of-2 textures. If this is supported, D3D11_TEXTURE_ADDRESS_MODE_WRAP can be used with such textures.

D3D11_FEATURE_DATA_D3D9_SHADOW_SUPPORT: Indicates support for comparison samplers in shader model 4.0 feature level 9_x shaders. This is used for depth testing in pixel shaders, enabling support for common techniques such as shadow mapping and stencils.

The following feature was added for feature level 9 devices starting in Windows 8.1:

D3D11_FEATURE_DATA_D3D9_SIMPLE_INSTANCING_SUPPORT: Indicates support for simple instancing features that might be available on DirectX 9-level hardware. Simple instancing means that all **InstanceDataStepRate** members of the **D3D11_INPUT_ELEMENT_DESC** structures used to define the input layout must be equal to 1. Devices that support feature level 9.3 or higher already include full support for instancing.

Optional floating-point precision support for shader programs

D3D11_FEATURE_DATA_SHADER_MIN_PRECISION_SUPPORT: The fields in this struct indicate the length of floating-point numbers when minimum precision is enabled, or 0 if only full 32-bit floating point precision is supported.

For feature level 9 devices, the minimum precision for the vertex shader can be different from the pixel shader. The precision for the vertex shader is indicated in the **AllOtherShaderStagesMinPrecision** field.

D3D11_FEATURE_DATA_DOUBLES: Feature level 11 devices can support double precision calculations within shader model 5.0 programs. Support for double precision calculations within the shader means that floats can be converted to doubles within the compute shader program, providing the benefit of higher precision computation within each shader pass. The double-precision numbers must be converted back to floats before being written to the output buffer. Note that double-precision division is not necessarily supported.

Additional capabilities for Direct3D 11.2

Direct3D 11.2 adds four new optional features that can be supported by Direct3D 11 devices. These features are in the [D3D11_FEATURE_DATA_D3D11_OPTIONS1](#) structure:

TiledResourcesTier: Indicates support for tiled resources, and indicates the tier level supported.

MinMaxFiltering: Indicates support for D3D11_FILTER_MINIMUM_* and D3D11_FILTER_MAXIMUM_* filtering options, which compare the filtering result to the minimum (or maximum) value. See [D3D11_FILTER](#).

ClearViewAlsoSupportsDepthOnlyFormats: Indicates support for clearing depth buffer resource views.

MapOnDefaultBuffers: Indicates support for mapping render target buffers created with the [D3D11_USAGE_DEFAULT](#) flag.

Tile based rendering

D3D11_FEATURE_DATA_ARCHITECTURE_INFO: Indicates whether the graphics device batches rendering commands, and performs tile-based rendering by default. This can be used as a hint for graphics engine optimization.

Optional features for development and debugging

D3D11_FEATURE_DATA_D3D11_OPTIONS::DiscardAPIsSeenByDriver: You can monitor this member during development to rule out legacy drivers on hardware where [DiscardView](#) and [DiscardResource](#) might have otherwise been beneficial.

D3D11_FEATURE_DATA_MARKER_SUPPORT: This is supported if the hardware and driver support data marking for GPU profiling.

Related topics

[Devices](#)

Resources

2/4/2021 • 2 minutes to read • [Edit Online](#)

Resources provide data to the pipeline and define what is rendered during your scene. Resources can be loaded from your game media or created dynamically at run time. Typically, resources include texture data, vertex data, and shader data. Most Direct3D applications create and destroy resources extensively throughout their lifespan. This section describes aspects of Direct3D 11 resources.

In this section

TOPIC	DESCRIPTION
Introduction to a Resource in Direct3D 11	This topic introduces Direct3D resources such as buffers and textures.
Types of Resources	This topic describes the types of resources from Direct3D 10, as well as new types in Direct3D 11 including structured buffers and writable textures and buffers.
Resource Limits	This topic contains a list of resources that Direct3D 11 supports (specifically feature level 11 or 9.x hardware).
Subresources	This topic describes texture subresources, or portions of a resource.
Buffers	Buffers contain data that is used for describing geometry, indexing geometry information, and shader constants. This section describes buffers that are used in Direct3D 11 and links to task-based documentation for common scenarios.
Textures	This section describes textures that are used in Direct3D 11 and links to task-based documentation for common scenarios.
Floating-point rules	Direct3D 11 supports several floating-point representations. All floating-point computations operate under a defined subset of the IEEE 754 32-bit single precision floating-point rules.
Tiled resources	Tiled resources can be thought of as large logical resources that use small amounts of physical memory.

Related topics

[Programming Guide for Direct3D 11](#)

Introduction to a Resource in Direct3D 11

11/2/2020 • 5 minutes to read • [Edit Online](#)

Resources are the building blocks of your scene. They contain most of the data that Direct3D uses to interpret and render your scene. Resources are areas in memory that can be accessed by the Direct3D pipeline. Resources contain the following types of data: geometry, textures, shader data. This topic introduces Direct3D resources such as buffers and textures.

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. Up to 128 resources can be active for each pipeline stage.

Direct3D guarantees to return zero for any resource that is accessed out of bounds.

The lifecycle of a Direct3D resource is:

- Create a resource using one of the create methods of the [ID3D11Device](#) interface.
- Bind a resource to the pipeline using a context and one of the set methods of the [ID3D11DeviceContext](#) interface.
- Deallocate a resource by calling the [Release](#) method of the resource interface.

This section contains the following topics:

- [Strong vs Weak Typing](#)
- [Resource Views](#)
- [Raw Views of Buffers](#)
- [Related topics](#)

Strong vs Weak Typing

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.

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. Resources created with a specific type cannot be reinterpreted using the view mechanism unless the resource was created with the D3D10_DDI_BIND_PRESENT flag. If D3D10_DDI_BIND_PRESENT is set render-target or shader resource views can be created on these resources using any of the fully typed members of the appropriate family, even if the original resource was created as fully typed.

In a type less resource, the data type is unknown when the resource is first created. The application must choose from the available type less formats (see [DXGI_FORMAT](#)). You must specify the size of the memory to allocate and whether the runtime will need to generate the subtextures in a mipmap. 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 [resource view](#). As the texture format remains flexible until the texture is bound to the pipeline, the resource is referred to as weakly typed storage. Weakly typed storage has the advantage that it can be reused or reinterpreted in another format as long as the number of components and the bit count of each component are the same in both formats.

A single resource can be bound to multiple pipeline stages as long as each has a unique view, which fully

qualifies the formats at each location. For example, a resource created with the format DXGI_FORMAT_R32G32B32A32_TYPELESS could be used as a DXGI_FORMAT_R32G32B32A32_FLOAT and a DXGI_FORMAT_R32G32B32A32_UINT at different locations in the pipeline simultaneously.

Resource Views

Resources can be stored in general purpose memory formats so that they can be shared by multiple pipeline stages. A pipeline stage interprets resource data using a view. A resource view is conceptually similar to casting the resource data so that it can be used in a particular context.

A view can be used with a typeless resource. That is, you can create a resource at compile time and declare the data type when the resource is bound to the pipeline. A view created for a typeless resource always has the same number of bits per component; the way the data is interpreted is dependent on the format specified. The format specified must be from the same family as the typeless format used when creating the resource. For example, a resource created with the R8G8B8A8_TYPELESS format cannot be viewed as a R32_FLOAT resource even though both formats may be the same size in memory.

A view also exposes other capabilities such as the ability to read back depth/stencil surfaces in a shader, generating a dynamic cubemap in a single pass, and rendering simultaneously to multiple slices of a volume.

RESOURCE INTERFACE	DESCRIPTION
ID3D11DepthStencilView	Access a texture resource during depth-stencil testing.
ID3D11RenderTargetView	Access a texture resource that is used as a render-target.
ID3D11ShaderResourceView	Access a shader resource such as a constant buffer, a texture buffer, a texture or a sampler.
ID3D11UnorderedAccessView	Access an unordered resource using a pixel shader or a compute shader.

Raw Views of Buffers

You can think of a raw buffer, which can also be called a [byte address buffer](#), as a bag of bits to which you want raw access, that is, a buffer that you can conveniently access through chunks of one to four 32-bit typeless address values. You indicate that you want raw access to a buffer (or, a raw view of a buffer) when you call one of the following methods to create a view to the buffer:

- To create a shader resource view (SRV) to the buffer, call [ID3D11Device::CreateShaderResourceView](#) with the flag [D3D11_BUFFEREX_SRV_FLAG_RAW](#). You specify this flag in the **Flags** member of the [D3D11_BUFFEREX_SRV](#) structure. You set D3D11_BUFFEREX_SRV in the **Buffer** member of the [D3D11_SHADER_RESOURCE_VIEW_DESC](#) structure to which the *pDesc* parameter of [ID3D11Device::CreateShaderResourceView](#) points. You also set the [D3D11_SRV_DIMENSION_BUFFEREX](#) value in the **ViewDimension** member of [D3D11_SHADER_RESOURCE_VIEW_DESC](#) to indicate that the SRV is a raw view.
- To create an unordered access view (UAV) to the buffer, call [ID3D11Device::CreateUnorderedAccessView](#) with the flag [D3D11_BUFFER_UAV_FLAG_RAW](#). You specify this flag in the **Flags** member of the [D3D11_BUFFER_UAV](#) structure. You set D3D11_BUFFER_UAV in the **Buffer** member of the [D3D11_UNORDERED_ACCESS_VIEW_DESC](#) structure to which the *pDesc* parameter of [ID3D11Device::CreateUnorderedAccessView](#) points. You also set the [D3D11_UAV_DIMENSION_BUFFER](#) value in the **ViewDimension** member of

`D3D11_UNORDERED_ACCESS_VIEW_DESC` to indicate that the UAV is a raw view.

You can use the HLSL `ByteAddressBuffer` and `RWByteAddressBuffer` object types when you work with raw buffers.

To create a raw view to a buffer, you must first call `ID3D11Device::CreateBuffer` with the `D3D11_RESOURCE_MISC_BUFFER_ALLOW_RAW_VIEWS` flag to create the underlying buffer resource. You specify this flag in the `MiscFlags` member of the `D3D11_BUFFER_DESC` structure to which the `pDesc` parameter of `ID3D11Device::CreateBuffer` points. You can't combine the `D3D11_RESOURCE_MISC_BUFFER_ALLOW_RAW_VIEWS` flag with `D3D11_RESOURCE_MISC_BUFFER_STRUCTURED`. Also, if you specify `D3D11_BIND_CONSTANT_BUFFER` in `BindFlags` of `D3D11_BUFFER_DESC`, you can't also specify `D3D11_RESOURCE_MISC_BUFFER_ALLOW_RAW_VIEWS` in `MiscFlags`. This is not a limitation of just raw views because constant buffers already have a constraint that they can't be combined with any other view.

Other than the preceding invalid cases, when you create a buffer with `D3D11_RESOURCE_MISC_BUFFER_ALLOW_RAW_VIEWS`, you aren't limited in functionality versus not setting `D3D11_RESOURCE_MISC_BUFFER_ALLOW_RAW_VIEWS`. That is, you can use such a buffer for non-raw access in any number of ways that are possible with Direct3D. If you specify the `D3D11_RESOURCE_MISC_BUFFER_ALLOW_RAW_VIEWS` flag, you only increase the available functionality.

Related topics

[Resources](#)

[New Resource Types](#)

Types of Resources

2/22/2020 • 2 minutes to read • [Edit Online](#)

Resources are areas in memory that can be accessed by the Direct3D pipeline, they are the building blocks of your scene. Resources contain many types of data such as geometry, textures or shader data that Direct3D uses to populate and render your scene. This topic describes the types of resources from Direct3D 10, as well as new types in Direct3D 11 including structured buffers and writable textures and buffers.

Differences between Direct3D 11 and Direct3D 10:

Direct3D 11 supports several new resource types including:

- [read-write buffers and textures](#)
- [structured buffers](#)
- [byte-address buffers](#)
- [append and consume buffers](#)
- [unordered access buffer or texture](#)

Direct3D 11.2 supports [tiled resources](#).

Both Direct3D 10 and Direct3D 11 support the [buffer](#) and [texture](#) types that were introduced in Direct3D 10.

Related topics

[Resources](#)

Resource Limits (Direct3D 11)

11/2/2020 • 4 minutes to read • [Edit Online](#)

This topic contains a list of resources that Direct3D 11 supports (specifically [feature level](#) 11 or 9.x hardware).

- [Resource limits for feature level 11 hardware](#)
- [Resource limits for feature level 9.x hardware](#)
- [Related topics](#)

Resource limits for feature level 11 hardware

All of these resource limits are defined as constants in D3d11.h.

RESOURCE	LIMIT
Number of elements in a constant buffer	D3D11_REQ_CONSTANT_BUFFER_ELEMENT_COUNT (4096)
Number of texels (independent of struct size) in a buffer	D3D11_REQ_BUFFER_RESOURCE_TEXEL_COUNT_2_TO_EXP (2 ⁷) texels
Texture1D U dimension	D3D11_REQ_TEXTURE1D_U_DIMENSION (16384)
Texture1DArray dimension	D3D11_REQ_TEXTURE1D_ARRAY_AXIS_DIMENSION (2048 array slices)
Texture2D U/V dimension	D3D11_REQ_TEXTURE2D_U_OR_V_DIMENSION (16384)
Texture2DArray dimension	D3D11_REQ_TEXTURE2D_ARRAY_AXIS_DIMENSION (2048 array slices)
Texture3D U/V/W dimension	D3D11_REQ_TEXTURE3D_U_V_OR_W_DIMENSION (2048)
TextureCube dimension	D3D11_REQ_TEXTURECUBE_DIMENSION (16384)
Resource size (in MB) for any of the preceding resources	min(max(128,0.25f * (amount of dedicated VRAM)), 2048) MB D3D11_REQ_RESOURCE_SIZE_IN_MEGABYTES_EXPRESSION_A_TERM (128) D3D11_REQ_RESOURCE_SIZE_IN_MEGABYTES_EXPRESSION_B_TERM (0.25f) D3D11_REQ_RESOURCE_SIZE_IN_MEGABYTES_EXPRESSION_C_TERM (2048)
Anisotropic filtering maxanisotropy	D3D11_REQ_MAXANISOTROPY (16)
Resource dimension addressable by filtering hardware	D3D11_REQ_FILTERING_HW_ADDRESSABLE_RESOURCE_DIMENSION (16384) per dimension

RESOURCE	LIMIT
Resource size (in MB) addressable by IA (input or vertex data) or VS/GS/PS (point sample)	max(128,0.25f * (amount of dedicated VRAM)) MB D3D11_REQ_RESOURCE_SIZE_IN_MEGABYTES_EXPRESSION_A_TERM (128) D3D11_REQ_RESOURCE_SIZE_IN_MEGABYTES_EXPRESSION_B_TERM (0.25f)
Total number of resource views per context (Each array counts as 1) (all view types have shared limit)	D3D11_REQ_RESOURCE_VIEW_COUNT_PER_DEVICE_2_TO_EXP (2)
Buffer structure size (multi-element)	D3D11_REQ_MULTI_ELEMENT_STRUCTURE_SIZE_IN_BYTES (2048 bytes)
Stream output size	Same as the number of texels in a buffer (see preceding)
Draw or DrawInstanced vertex count (including instancing)	D3D11_REQ_DRAW_VERTEX_COUNT_2_TO_EXP (2)
DrawIndexed[Instanced]() vertex count (including instancing)	D3D11_REQ_DRAWINDEXED_INDEX_COUNT_2_TO_EXP (2)
GS invocation output data (components * vertices)	D3D11_GS_MAX_OUTPUT_VERTEX_COUNT_ACROSS_INSTANCES (1024)
Total number of sampler objects per context	D3D11_REQ_SAMPLER_OBJECT_COUNT_PER_DEVICE (4096)
Total number of viewport/scissor objects per pipeline	D3D11_VIEWPORT_AND_SCISSORRECT_OBJECT_COUNT_PER_PIPELINE (16)
Total number of clip/cull distances per vertex	D3D11_CLIP_OR_CULL_DISTANCE_COUNT (8)
Total number of blend objects per context	D3D11_REQ_BLEND_OBJECT_COUNT_PER_DEVICE (4096)
Total number of depth/stencil objects per context	D3D11_REQ_DEPTH_STENCIL_OBJECT_COUNT_PER_DEVICE (4096)
Total number of rasterizer state objects per context	D3D11_REQ_RASTERIZER_OBJECT_COUNT_PER_DEVICE (4096)
Maximum per-pixel sample count during multisampling	D3D11_MAX_MULTISAMPLE_SAMPLE_COUNT (32)
Shader resource vertex-element count (four 32-bit components)	D3D11_STANDARD_VERTEX_ELEMENT_COUNT (32)
Common-shader core (four 32-bit components) temp-register count (r# + indexable x#[n])	D3D11_COMMONSHADER_TEMP_REGISTER_COUNT (4096)
Common-shader core constant-buffer slots	D3D11_COMMONSHADER_CONSTANT_BUFFER_HW_SLOT_COUNT (15) (+1 set aside for an immediate constant buffer in shaders)
Common-shader core input-resource slots	D3D11_COMMONSHADER_INPUT_RESOURCE_REGISTER_COUNT (128)
Common-shader core sampler slots	D3D11_COMMONSHADER_SAMPLER_SLOT_COUNT (16)

RESOURCE	LIMIT
Common-shader core subroutine-nesting limit	D3D11_COMMONSHADER_SUBROUTINE_NESTING_LIMIT (32)
Common-shader core flow-control nesting limit	D3D11_COMMONSHADER_FLOWCONTROL_NESTING_LIMIT (64)
Vertex shader input-register count (four 32-bit components)	D3D11_VS_INPUT_REGISTER_COUNT (32)
Vertex shader output-register count (four 32-bit components)	D3D11_VS_OUTPUT_REGISTER_COUNT (32)
Geometry shader input-register count (four 32-bit components)	D3D11_GS_INPUT_REGISTER_COUNT (32)
Geometry shader output-register count (four 32-bit components)	D3D11_GS_OUTPUT_REGISTER_COUNT (32)
Pixel shader input-register count (four 32-bit components)	D3D11_PS_INPUT_REGISTER_COUNT (32)
Pixel shader output slots	8
Pixel shader output depth register count(one 32-bit component)	D3D11_PS_OUTPUT_DEPTH_REGISTER_COUNT (1)
Input assembler index input resource slots	D3D11_IA_INDEX_INPUT_RESOURCE_SLOT_COUNT (1)
Input assembler vertex input resource slots	D3D11_IA_VERTEX_INPUT_RESOURCE_SLOT_COUNT (32)
Hull shader control point input-register count (four 32-bit components)	D3D11_HS_CONTROL_POINT_PHASE_INPUT_REGISTER_COUNT (32)
Hull shader number of input control points	D3D11_HS_CONTROL_POINT_REGISTER_COMPONENT_BIT_COUNT (32)
Hull shader control point output-register count (four 32-bit components)	D3D11_HS_CONTROL_POINT_PHASE_OUTPUT_REGISTER_COUNT (32)
Hull shader number of output control points	D3D11_HS_CONTROL_POINT_REGISTER_COMPONENT_BIT_COUNT (32)
Hull shader patch constant output-register count (four 32-bit components)	D3D11_HS_OUTPUT_PATCH_CONSTANT_REGISTER_COUNT (32)
Domain shader control point input-register count (four 32-bit components)	D3D11_DS_INPUT_CONTROL_POINT_REGISTER_COUNT (32)
Domain shader number of input control points	D3D11_DS_INPUT_CONTROL_POINT_REGISTER_COMPONENT_BIT_COUNT (32)
Domain shader patch constant input-register count (four 32-bit components)	D3D11_DS_INPUT_PATCH_CONSTANT_REGISTER_COUNT (32)

RESOURCE	LIMIT
Domain shader tessellated vertex output-register count (four 32-bit components)	D3D11_DS_OUTPUT_REGISTER_COUNT (32)
Compute shader unordered access view (UAV) slots	D3D11_PS_CS_UAV_REGISTER_COUNT (8) ⁴
Resource tile size in bytes	D3D11_2_TILED_RESOURCE_TILE_SIZE_IN_BYTES (65536)

An app can try to allocate more memory for a resource than the maximum resource size specifies. That is, the Direct3D 11 runtime allows these memory allocation attempts in the event that the hardware might support them. However, the Direct3D 11 runtime only guarantees that allocations within the maximum resource size are supported by all [feature level](#) 11 hardware. If an app tries to allocate memory for a resource within the maximum resource size, the runtime fails the attempt only if the operating system runs out of resources. If an app tries to allocate memory for a resource above the maximum resource size, the runtime can fail the attempt because either the operating system is overextended or the hardware does not support allocations above the maximum resource size. The debug layer only checks

D3D11_REQ_RESOURCE_SIZE_IN_MEGABYTES_EXPRESSION_A_TERM (128) MB.

The pixel shader output slots are shared between pixel output registers (four 32-bit components) and UAVs.

The total number of components for all hull shader to domain shader control points is limited to 3968, which is 128 less than the maximum control points times the maximum control point registers times four components.

⁴For compute shader profiles CS_4_0 and CS_4_1 there is only 1 UAV available. For more information about shader profiles, see [Shader Model 5](#).

Resource limits for feature level 9.x hardware

All of these 9.x [feature level](#) resource limits are defined as constants in D3dcommon.h.

RESOURCE	LIMIT
Feature level 9_1 texture1D U dimension	D3D_FL9_1_REQ_TEXTURE1D_U_DIMENSION (2048)
Feature level 9_3 texture1D U dimension	D3D_FL9_3_REQ_TEXTURE1D_U_DIMENSION (4096)
Feature level 9_1 texture2D U/V dimension	D3D_FL9_1_REQ_TEXTURE2D_U_OR_V_DIMENSION (2048)
Feature level 9_3 texture2D U/V dimension	D3D_FL9_3_REQ_TEXTURE2D_U_OR_V_DIMENSION (4096)
Feature level 9_1 texture3D U/V/W dimension	D3D_FL9_1_REQ_TEXTURE3D_U_V_OR_W_DIMENSION (256)
Feature level 9_1 textureCube dimension	D3D_FL9_1_REQ_TEXTURECUBE_DIMENSION (512)
Feature level 9_3 textureCube dimension	D3D_FL9_3_REQ_TEXTURECUBE_DIMENSION (4096)
Feature level 9_1 anisotropic filtering maxanisotropy default	D3D_FL9_1_DEFAULT_MAX_ANISOTROPY (2)
Feature level 9_1 maximum input assembler primitives	D3D_FL9_1_IA_PRIMITIVE_MAX_COUNT (65535)
Feature level 9_2 maximum input assembler primitives	D3D_FL9_2_IA_PRIMITIVE_MAX_COUNT (1048575)

RESOURCE	LIMIT
Feature level 9_1 simultaneous render targets	D3D_FL9_1_SIMULTANEOUS_RENDER_TARGET_COUNT (1)
Feature level 9_3 simultaneous render targets	D3D_FL9_3_SIMULTANEOUS_RENDER_TARGET_COUNT (4)
Feature level 9_1 maximum texture repeat	D3D_FL9_1_MAX_TEXTURE_REPEAT (128)
Feature level 9_2 maximum texture repeat	D3D_FL9_2_MAX_TEXTURE_REPEAT (2048)
Feature level 9_3 maximum texture repeat	D3D_FL9_3_MAX_TEXTURE_REPEAT (8192)

Related topics

[Resources](#)

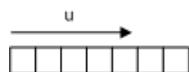
Subresources (Direct3D 11 Graphics)

2/4/2021 • 2 minutes to read • [Edit Online](#)

This topic describes texture subresources, or portions of a resource.

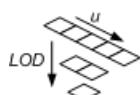
Direct3D can reference an entire resource or it can reference subsets of a resource. The term subresource refers to a subset of a resource.

A buffer is defined as a single subresource. Textures are a little more complicated because there are several different texture types (1D, 2D, etc.) some of which support mipmap levels and/or texture arrays. Beginning with the simplest case, a 1D texture is defined as a single subresource, as shown in the following illustration.

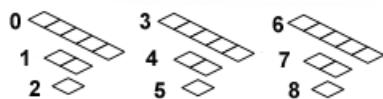


This means that the array of texels that make up a 1D texture are contained in a single subresource.

If you expand a 1D texture with three mipmap levels, it can be visualized like the following illustration.



Think of this as a single texture that is made up of three subresources. A subresource can be indexed using the level-of-detail (LOD) for a single texture. When using an array of textures, accessing a particular subresource requires both the LOD and the particular texture. Alternately, the API combines these two pieces of information into a single zero-based subresource index, as shown in the following illustration.



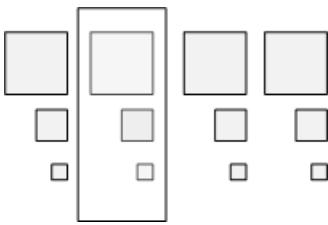
Selecting Subresources

Some APIs access an entire resource (for example the [ID3D11DeviceContext::CopyResource](#) method), others access a portion of a resource (for example the [ID3D11DeviceContext::UpdateSubresource](#) method or the [ID3D11DeviceContext::CopySubresourceRegion](#) method). The methods that access a portion of a resource generally use a view description (such as the [D3D11_TEX2D_ARRAY_DSV](#) structure) to specify the subresources to access.

The illustrations in the following sections show the terms used by a view description when accessing an array of textures.

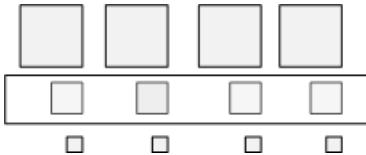
Array Slice

Given an array of textures, each texture with mipmaps, an *array slice* (represented by the white rectangle) includes one texture and all of its subresources, as shown in the following illustration.



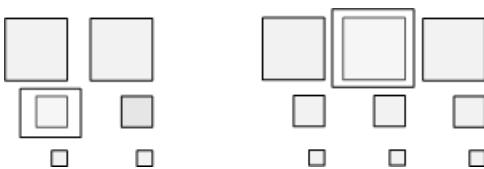
Mip Slice

A *mip slice* (represented by the white rectangle) includes one mipmap level for every texture in an array, as shown in the following illustration.



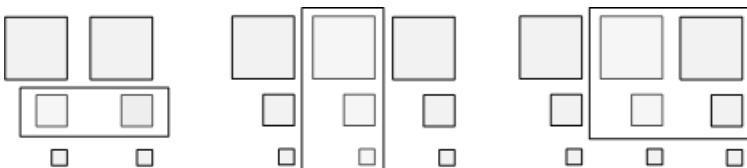
Selecting a Single Subresource

You can use these two types of slices to choose a single subresource, as shown in the following illustration.



Selecting Multiple Subresources

Or you can use these two types of slices with the number of mipmap levels and/or number of textures, to choose multiple subresources, as shown in the following illustration.



NOTE

A [render-target view](#) can only use a single subresource or mip slice and cannot include subresources from more than one mip slice. That is, every texture in a render-target view must be the same size. A [shader-resource view](#) can select any rectangular region of subresources, as shown in the figure.

Related topics

[Resources](#)

Buffers

2/22/2020 • 2 minutes to read • [Edit Online](#)

Buffers contain data that is used for describing geometry, indexing geometry information, and shader constants. This section describes buffers that are used in Direct3D 11 and links to task-based documentation for common scenarios.

In this section

TOPIC	DESCRIPTION
Introduction to Buffers in Direct3D 11	A buffer resource is a collection of fully typed data grouped into elements. You can use buffers to store a wide variety of data, including position vectors, normal vectors, texture coordinates in a vertex buffer, indexes in an index buffer, or device state. A buffer element is made up of 1 to 4 components. Buffer elements can include packed data values (like R8G8B8A8 surface values), single 8-bit integers, or four 32-bit floating point values.

Related topics

[How to: Create a Constant Buffer](#)

[How to: Create an Index Buffer](#)

[How to: Create a Vertex Buffer](#)

[Resources](#)

Introduction to Buffers in Direct3D 11

11/2/2020 • 3 minutes to read • [Edit Online](#)

A buffer resource is a collection of fully typed data grouped into elements. You can use buffers to store a wide variety of data, including position vectors, normal vectors, texture coordinates in a vertex buffer, indexes in an index buffer, or device state. A buffer element is made up of 1 to 4 components. Buffer elements can include packed data values (like R8G8B8A8 surface values), single 8-bit integers, or four 32-bit floating point values.

A buffer is created as an unstructured resource. Because it is unstructured, a buffer cannot contain any mipmap levels, it cannot get filtered when read, and it cannot be multisampled.

Buffer Types

The following are the buffer resource types supported by Direct3D 11. All buffer types are encapsulated by the [ID3D11Buffer](#) interface.

- [Vertex Buffer](#)
- [Index Buffer](#)
- [Constant Buffer](#)

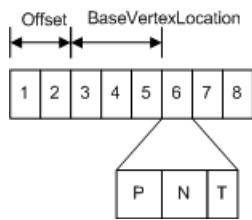
Vertex Buffer

A vertex buffer contains the vertex data used to define your geometry. Vertex data includes position coordinates, color data, texture coordinate data, normal data, and so on.

The simplest example of a vertex buffer is one that only contains position data. It can be visualized like the following illustration.



More often, a vertex buffer contains all the data needed to fully specify 3D vertices. An example of this could be a vertex buffer that contains per-vertex position, normal and texture coordinates. This data is usually organized as sets of per-vertex elements, as shown in the following illustration.



This vertex buffer contains per-vertex data; each vertex stores three elements (position, normal, and texture coordinates). The position and normal are each typically specified using three 32-bit floats (DXGI_FORMAT_R32G32B32_FLOAT) and the texture coordinates using two 32-bit floats (DXGI_FORMAT_R32G32_FLOAT).

To access data from a vertex buffer you need to know which vertex to access, plus the following additional buffer parameters:

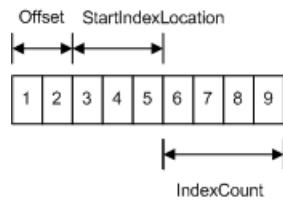
- Offset - the number of bytes from the start of the buffer to the data for the first vertex. You can specify the offset using the [ID3D11DeviceContext::IASetVertexBuffers](#) method.
- BaseVertexLocation - the number of bytes from the offset to the first vertex used by the appropriate draw call.

Before you create a vertex buffer, you need to define its layout by creating an [ID3D11InputLayout](#) interface; this is done by calling the [ID3D11Device::CreateInputLayout](#) method. After the input-layout object is created, you can bind it to the input-assembler stage by calling the [ID3D11DeviceContext::IASetInputLayout](#).

To create a vertex buffer, call [ID3D11Device::CreateBuffer](#).

Index Buffer

Index buffers contain integer offsets into vertex buffers and are used to render primitives more efficiently. An index buffer contains a sequential set of 16-bit or 32-bit indices; each index is used to identify a vertex in a vertex buffer. An index buffer can be visualized like the following illustration.



The sequential indices stored in an index buffer are located with the following parameters:

- Offset - the number of bytes from the base address of the index buffer. The offset is supplied to the [ID3D11DeviceContext::IASetIndexBuffer](#) method.
- StartIndexLocation - specifies the first index buffer element from the base address and the offset provided in [IASetIndexBuffer](#). The start location is supplied to the [ID3D11DeviceContext::DrawIndexed](#) or [ID3D11DeviceContext::DrawIndexedInstanced](#) method and represents the first index to render.
- IndexCount - the number of indices to render. The number is supplied to the [DrawIndexed](#) method

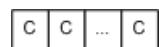
Start of Index Buffer = Index Buffer Base Address + Offset (bytes) + StartIndexLocation * ElementSize (bytes);

In this calculation, ElementSize is the size of each index buffer element, which is either two or four bytes.

To create an index buffer, call [ID3D11Device::CreateBuffer](#).

Constant Buffer

A constant buffer allows you to efficiently supply shader constants data to the pipeline. You can use a constant buffer to store the results of the stream-output stage. Conceptually, a constant buffer looks just like a single-element vertex buffer, as shown in the following illustration.



Each element stores a 1-to-4 component constant, determined by the format of the data stored. To create a shader-constant buffer, call [ID3D11Device::CreateBuffer](#) and specify the [D3D11_BIND_CONSTANT_BUFFER](#) member of the [D3D11_BIND_FLAG](#) enumerated type.

A constant buffer can only use a single bind flag ([D3D11_BIND_CONSTANT_BUFFER](#)), which cannot be combined with any other bind flag. To bind a shader-constant buffer to the pipeline, call one of the following methods: [ID3D11DeviceContext::GSSetConstantBuffers](#), [ID3D11DeviceContext::PSSetConstantBuffers](#), or [ID3D11DeviceContext::VSSetConstantBuffers](#).

To read a shader-constant buffer from a shader, use a HLSL load function (for example, [Load](#)). Each shader stage allows up to 15 shader-constant buffers; each buffer can hold up to 4096 constants.

Related topics

[Buffers](#)

Textures

2/22/2020 • 2 minutes to read • [Edit Online](#)

A texture stores texel information. This section describes textures that are used in Direct3D 11 and links to task-based documentation for common scenarios.

In this section

TOPIC	DESCRIPTION
Introduction To Textures in Direct3D 11	A texture resource is a structured collection of data designed to store texels. A texel represents the smallest unit of a texture that can be read or written to by the pipeline. 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. Each texel contains 1 to 4 components, arranged in one of the DXGI formats defined by the DXGI_FORMAT enumeration.
Texture Block Compression in Direct3D 11	Block Compression (BC) support for textures has been extended in Direct3D 11 to include the BC6H and BC7 algorithms. BC6H supports high-dynamic range color source data, and BC7 provides better-than-average quality compression with less artifacts for standard RGB source data.

Related topics

[How to: Create a Texture](#)

[How to: Initialize a Texture Programmatically](#)

[How to: Initialize a Texture From a File](#)

[Resources](#)

Introduction To Textures in Direct3D 11

2/22/2020 • 4 minutes to read • [Edit Online](#)

A texture resource is a structured collection of data designed to store texels. A texel represents the smallest unit of a texture that can be read or written to by the pipeline. 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. Each texel contains 1 to 4 components, arranged in one of the DXGI formats defined by the DXGI_FORMAT enumeration.

Textures are created as a structured resource with a known size. However, each texture may be typed or typeless when the resource is created as long as the type is fully specified using a view when the texture is bound to the pipeline.

Texture Types

There are several types of textures: 1D, 2D, 3D, each of which can be created with or without mipmaps. Direct3D 11 also supports texture arrays and multisampled textures.

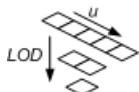
- [1D Textures](#)
- [1D Texture Arrays](#)
- [2D Textures and 2D Texture Arrays](#)
- [3D Textures](#)

1D Textures

A 1D texture in its simplest form contains texture data that can be addressed with a single texture coordinate; it can be visualized as an array of texels, as shown in the following illustration. 1D textures are represented by the [ID3D11Texture1D](#) interface.



Each texel contains a number of color components depending on the format of the data stored. Adding more complexity, you can create a 1D texture with mipmap levels, as shown in the following illustration.

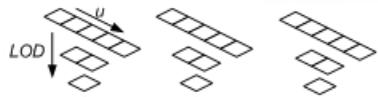


A mipmap level is a texture that is a power-of-two smaller than the level above it. The topmost level contains the most detail, each subsequent level is smaller. For a 1D mipmap, the smallest level contains one texel.

Furthermore, MIP levels always reduce down to 1:1. When mipmaps are generated for an odd sized texture, the next lower level is always even size (except when the lowest level reaches 1). For example, the diagram illustrates a 5x1 texture whose next lowest level is a 2x1 texture, whose next (and last) mip level is a 1x1 sized texture. The levels are identified by an index called a LOD (level-of-detail) which is used to access the smaller texture when rendering geometry that is not as close to the camera.

1D Texture Arrays

Direct3D 11 also supports arrays of textures. A 1D texture array is also represented by the [ID3D11Texture1D](#) interface. An array of 1D textures looks conceptually like the following illustration.



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 first layer). Each texture also contains a 3 layer mipmap.

All texture arrays in Direct3D are a homogeneous 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.

2D Textures and 2D Texture Arrays

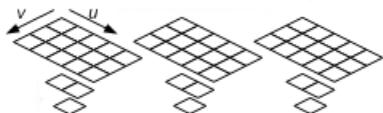
A Texture2D resource contains a 2D grid of texels. Each texel is addressable by a u, v vector. Since it is a texture resource, it may contain mipmap levels, and subresources. 2D textures are represented by the

[ID3D11Texture2D](#) interface. A fully populated 2D texture resource looks like the following illustration.



This texture resource contains a single 3x5 texture with three mipmap levels.

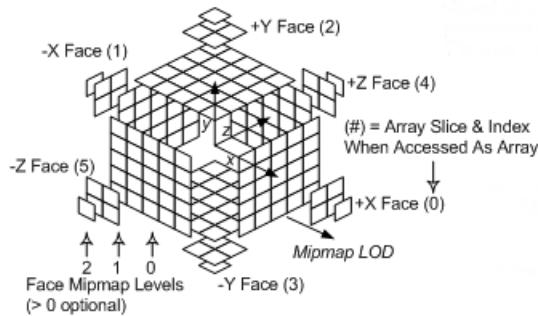
A 2D texture array resource is a homogeneous array of 2D textures; that is, each texture has the same data format and dimensions (including mipmap levels). A 2D texture array is also represented by the [ID3D11Texture2D](#) interface. It has a similar layout as the 1D texture array except that the textures now contain 2D data, as shown in the following illustration.



This texture array contains three textures; each texture is 3x5 with two mipmap levels.

Using a 2D Texture Array as a Texture Cube

A texture cube is a 2D texture array that contains 6 textures, one for each face of the cube. A fully populated texture cube looks like the following illustration.



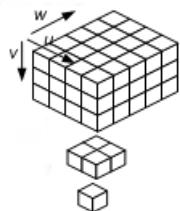
A 2D texture array that contains 6 textures may be read from within shaders with the cube map intrinsic functions, after they are bound to the pipeline with a cube-texture view. Texture cubes are addressed from the shader with a 3D vector pointing out from the center of the texture cube.

NOTE

Devices that you create with **10_1 feature level** and above can support arrays of texture cubes in which the number of textures is equal to the number of texture cubes in an array times six. Devices that you create with **10_0 feature level** support only a single texture cube of six faces. Also, Direct3D 11 does not support partial cubemaps.

3D Textures

A 3D texture resource (also known as a volume texture) contains a 3D volume of texels. Because it is a texture resource, it may contain mipmap levels. 3D textures are represented by the **ID3D11Texture3D** interface. A fully populated 3D texture looks like the following illustration.



When a 3D texture mipmap slice is bound as a render target output (with a render-target view), the 3D texture behaves identically to a 2D texture array with n slices. The particular render slice 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 a 3D texture array; therefore a 3D texture subresource is a single mipmap level.

Related topics

[Textures](#)

Texture Block Compression in Direct3D 11

11/2/2020 • 3 minutes to read • [Edit Online](#)

Block Compression (BC) support for textures has been extended in Direct3D 11 to include the BC6H and BC7 algorithms. BC6H supports high-dynamic range color source data, and BC7 provides better-than-average quality compression with less artifacts for standard RGB source data.

For more specific information about block compression algorithm support prior to Direct3D 11, including support for the BC1 through BC5 formats, see [Block Compression \(Direct3D 10\)](#).

Note about File Formats: The BC6H and BC7 texture compression formats use the DDS file format for storing the compressed texture data. For more information, see the [Programming Guide for DDS](#) for details.

Block Compression Formats Supported in Direct3D 11

SOURCE DATA	MINIMUM REQUIRED DATA COMPRESSION RESOLUTION	RECOMMENDED FORMAT	MINIMUM SUPPORTED FEATURE LEVEL
Three-channel color with alpha channel	Three color channels (5 bits:6 bits:5 bits), with 0 or 1 bit(s) of alpha	BC1	Direct3D 9.1
Three-channel color with alpha channel	Three color channels (5 bits:6 bits:5 bits), with 4 bits of alpha	BC2	Direct3D 9.1
Three-channel color with alpha channel	Three color channels (5 bits:6 bits:5 bits) with 8 bits of alpha	BC3	Direct3D 9.1
One-channel color	One color channel (8 bits)	BC4	Direct3D 10
Two-channel color	Two color channels (8 bits:8 bits)	BC5	Direct3D 10
Three-channel high dynamic range (HDR) color	Three color channels (16 bits:16 bits:16 bits) in "half" floating point*	BC6H	Direct3D 11
Three-channel color, alpha channel optional	Three color channels (4 to 7 bits per channel) with 0 to 8 bits of alpha	BC7	Direct3D 11

*"Half" floating point is a 16 bit value that consists of an optional sign bit, a 5 bit biased exponent, and a 10 or 11 bit mantissa.

BC1, BC2, and BC3 Formats

The BC1, BC2, and BC3 formats are equivalent to the Direct3D 9 DXTn texture compression formats, and are the same as the corresponding Direct3D 10 BC1, BC2, and BC3 formats. Support for these three formats is required by all feature levels (D3D_FEATURE_LEVEL_9_1, D3D_FEATURE_LEVEL_9_2, D3D_FEATURE_LEVEL_9_3, D3D_FEATURE_LEVEL_10_0, D3D_FEATURE_LEVEL_10_1, and D3D_FEATURE_LEVEL_11_0).

BLOCK COMPRESSION FORMAT	DXGI FORMAT	DIRECT3D 9 EQUIVALENT FORMAT	BYTES PER 4X4 PIXEL BLOCK
BC1	DXGI_FORMAT_BC1_UNORM, DXGI_FORMAT_BC1_UNORM_SRGB, DXGI_FORMAT_BC1_TYPELESS	D3DFMT_DXT1, FourCC="DXT1"	8
BC2	DXGI_FORMAT_BC2_UNORM, DXGI_FORMAT_BC2_UNORM_SRGB, DXGI_FORMAT_BC2_TYPELESS	D3DFMT_DXT2*, FourCC="DXT2", D3DFMT_DXT3, FourCC="DXT3"	16
BC3	DXGI_FORMAT_BC3_UNORM, DXGI_FORMAT_BC3_UNORM_SRGB, DXGI_FORMAT_BC3_TYPELESS	D3DFMT_DXT4*, FourCC="DXT4", D3DFMT_DXT5, FourCC="DXT5"	16

*These compression schemes (DXT2 and DXT4) make no distinction between the Direct3D 9 pre-multiplied alpha formats and the standard alpha formats. This distinction must be handled by the programmable shaders at render time.

BC4 and BC5 Formats

BLOCK COMPRESSION FORMAT	DXGI FORMAT	DIRECT3D 9 EQUIVALENT FORMAT	BYTES PER 4X4 PIXEL BLOCK
BC4	DXGI_FORMAT_BC4_UNORM, DXGI_FORMAT_BC4_SNORM, DXGI_FORMAT_BC4_TYPELESS	FourCC="ATI1"	8
BC5	DXGI_FORMAT_BC5_UNORM, DXGI_FORMAT_BC5_SNORM, DXGI_FORMAT_BC5_TYPELESS	FourCC="ATI2"	16

BC6H Format

For more detailed information about this format, see the [BC6H Format](#) documentation.

BLOCK COMPRESSION FORMAT	DXGI FORMAT	DIRECT3D 9 EQUIVALENT FORMAT	BYTES PER 4X4 PIXEL BLOCK

BLOCK COMPRESSION FORMAT	DXGI FORMAT	DIRECT3D 9 EQUIVALENT FORMAT	BYTES PER 4X4 PIXEL BLOCK
BC6H	DXGI_FORMAT_BC6H_UF16, DXGI_FORMAT_BC6H_SF16, DXGI_FORMAT_BC6H_TYPELESS	N/A	16

The BC6H format can select different encoding modes for each 4x4 pixel block. A total of 14 different encoding modes are available, each with slightly different trade-offs in the resulting visual quality of the displayed texture. The choice of modes allows for fast decoding by the hardware with the quality level selected or adapted according to the source content, but it also greatly increases the complexity of the search space.

BC7 Format

For more detailed information about this format, see the [BC7 Format](#) documentation.

BLOCK COMPRESSION FORMAT	DXGI FORMAT	DIRECT3D 9 EQUIVALENT FORMAT	BYTES PER 4X4 PIXEL BLOCK
BC7	DXGI_FORMAT_BC7_UNORM, DXGI_FORMAT_BC7_UNORM_SRGB, DXGI_FORMAT_BC7_TYPELESS	N/A	16

The BC7 format can select different encoding modes for each 4x4 pixel block. A total of 8 different encoding modes are available, each with slightly different trade-offs in the resulting visual quality of the displayed texture. The choice of modes allows for fast decoding by the hardware with the quality level selected or adapted according to the source content, but it also greatly increases the complexity of the search space.

In this section

TOPIC	DESCRIPTION
BC6H Format	The BC6H format is a texture compression format designed to support high-dynamic range (HDR) color spaces in source data.
BC7 Format	The BC7 format is a texture compression format used for high-quality compression of RGB and RGBA data.
BC7 Format Mode Reference	This documentation contains a list of the 8 block modes and bit allocations for BC7 texture compression format blocks.

Related topics

[Block Compression \(Direct3D 10\)](#)

[Textures](#)

BC6H Format

11/2/2020 • 11 minutes to read • [Edit Online](#)

The BC6H format is a texture compression format designed to support high-dynamic range (HDR) color spaces in source data.

- [About BC6H/DXGI_FORMAT_BC6H](#)
- [BC6H Implementation](#)
- [Decoding the BC6H Format](#)
- [BC6H Compressed Endpoint Format](#)
- [Sign Extension for Endpoint Values](#)
- [Transform Inversion for Endpoint Values](#)
- [Unquantization of Color Endpoints](#)
- [Related topics](#)

About BC6H/DXGI_FORMAT_BC6H

The BC6H format provides high-quality compression for images that use three HDR color channels, with a 16-bit value for each color channel of the value (16:16:16). There is no support for an alpha channel.

BC6H is specified by the following DXGI_FORMAT enumeration values:

- [DXGI_FORMAT_BC6H_TYPELESS](#).
- [DXGI_FORMAT_BC6H_UF16](#). This BC6H format does not use a sign bit in the 16-bit floating point color channel values.
- [DXGI_FORMAT_BC6H_SF16](#). This BC6H format uses a sign bit in the 16-bit floating point color channel values.

NOTE

The 16 bit floating point format for color channels is often referred to as a "half" floating point format. This format has the following bit layout:

UF16 (unsigned float)	5 exponent bits + 11 mantissa bits
SF16 (signed float)	1 sign bit + 5 exponent bits + 10 mantissa bits

The BC6H format can be used for [Texture2D](#) (including arrays), [Texture3D](#), or [TextureCube](#) (including arrays) texture resources. Similarly, this format applies to any MIP-map surfaces associated with these resources.

BC6H uses a fixed block size of 16 bytes (128 bits) and a fixed tile size of 4x4 texels. As with previous BC formats, texture images larger than the supported tile size (4x4) are compressed by using multiple blocks. This addressing identity applies also to three-dimensional images, MIP-maps, cubemaps, and texture arrays. All image tiles must be of the same format.

Some important notes about the BC6H format:

- BC6H supports floating point denormalization, but does not support INF (infinity) and NaN (not a number). The exception is the signed mode of BC6H (DXGI_FORMAT_BC6H_SF16), which supports -INF (negative infinity). Note that this support for -INF is merely an artifact of the format itself, and is not specifically supported by encoders for this format. In general, when encoders encounter INF (positive or negative) or NaN input data, they should convert that data to the maximum allowable non-INF representation value, and map NaN to 0 prior to compression.
- BC6H does not support an alpha channel.
- The BC6H decoder performs decompression before it performs texture filtering.
- BC6H decompression must be bit accurate; that is, the hardware must return results that are identical to the decoder described in this documentation.

BC6H Implementation

A BC6H block consists of mode bits, compressed endpoints, compressed indices, and an optional partition index. This format specifies 14 different modes.

An endpoint color is stored as an RGB triplet. BC6H defines a palette of colors on an approximate line across a number of defined color endpoints. Also, depending on the mode, a tile can be divided into two regions or treated as a single region, where a two-region tile has a separate set of color endpoints for each region. BC6H stores one palette index per texel.

In the two-region case, there are 32 possible partitions.

Decoding the BC6H Format

The pseudocode below shows the steps to decompress the pixel at (x,y) given the 16 byte BC6H block.

```

decompress_bc6h(x, y, block)
{
    mode = extract_mode(block);
    endpoints;
    index;

    if(mode.type == ONE)
    {
        endpoints = extract_compressed_endpoints(mode, block);
        index = extract_index_ONE(x, y, block);
    }
    else //mode.type == TWO
    {
        partition = extract_partition(block);
        region = get_region(partition, x, y);
        endpoints = extract_compressed_endpoints(mode, region, block);
        index = extract_index_TWO(x, y, partition, block);
    }

    unquantize(endpoints);
    color = interpolate(index, endpoints);
    finish_unquantize(color);
}

```

The following table contains the bit count and values for each of the 14 possible formats for BC6H blocks.

MODE	PARTITION INDICES	PARTITION	COLOR ENDPOINTS	MODE BITS
1	46 bits	5 bits	75 bits (10.555, 10.555, 10.555)	2 bits (00)

MODE	PARTITION INDICES	PARTITION	COLOR ENDPOINTS	MODE BITS
2	46 bits	5 bits	75 bits (7666, 7666, 7666)	2 bits (01)
3	46 bits	5 bits	72 bits (11.555, 11.444, 11.444)	5 bits (00010)
4	46 bits	5 bits	72 bits (11.444, 11.555, 11.444)	5 bits (00110)
5	46 bits	5 bits	72 bits (11.444, 11.444, 11.555)	5 bits (01010)
6	46 bits	5 bits	72 bits (9555, 9555, 9555)	5 bits (01110)
7	46 bits	5 bits	72 bits (8666, 8555, 8555)	5 bits (10010)
8	46 bits	5 bits	72 bits (8555, 8666, 8555)	5 bits (10110)
9	46 bits	5 bits	72 bits (8555, 8555, 8666)	5 bits (11010)
10	46 bits	5 bits	72 bits (6666, 6666, 6666)	5 bits (11110)
11	63 bits	0 bits	60 bits (10.10, 10.10, 10.10)	5 bits (00011)
12	63 bits	0 bits	60 bits (11.9, 11.9, 11.9)	5 bits (00111)
13	63 bits	0 bits	60 bits (12.8, 12.8, 12.8)	5 bits (01011)
14	63 bits	0 bits	60 bits (16.4, 16.4, 16.4)	5 bits (01111)

Each format in this table can be uniquely identified by the mode bits. The first ten modes are used for two-region tiles, and the mode bit field can be either two or five bits long. These blocks also have fields for the compressed color endpoints (72 or 75 bits), the partition (5 bits), and the partition indices (46 bits).

For the compressed color endpoints, the values in the preceding table note the precision of the stored RGB endpoints, and the number of bits used for each color value. For example, mode 3 specifies a color endpoint precision level of 11, and the number of bits used to store the delta values of the transformed endpoints for the red, blue and green colors (5, 4, and 4 respectively). Mode 10 does not use delta compression, and instead stores all four color endpoints explicitly.

The last four block modes are used for one-region tiles, where the mode field is 5 bits. These blocks have fields for the endpoints (60 bits) and the compressed indices (63 bits). Mode 11 (like Mode 10) does not use delta compression, and instead stores both color endpoints explicitly.

Modes 10011, 10111, 11011, and 11111 (not shown) are reserved. Do not use these in your encoder. If the hardware is passed blocks with one of these modes specified, the resulting decompressed block must contain all zeroes in all channels except for the alpha channel.

For BC6H, the alpha channel must always return 1.0 regardless of the mode.

BC6H Partition Set

There are 32 possible partition sets for a two-region tile, and which are defined in the table below. Each 4x4 block represents a single shape.

0, 0, 1, 1,	0, 0, 0, 1,	0, 1, 1, 1,	0, 0, 0, 1,
0, 0, 1, 1,	0, 0, 0, 1,	0, 1, 1, 1,	0, 0, 1, 1,
0, 0, 1, 1,	0, 0, 0, 1,	0, 1, 1, 1,	0, 0, 1, 1,
0, 0, 1, 1,	0, 0, 0, 1,	0, 1, 1, 1,	0, 1, 1, 1,
0, 0, 0, 0,	0, 0, 1, 1,	0, 0, 0, 1,	0, 0, 0, 0,
0, 0, 0, 1,	0, 1, 1, 1,	0, 0, 1, 1,	0, 0, 0, 1,
0, 0, 0, 1,	0, 1, 1, 1,	0, 1, 1, 1,	0, 0, 1, 1,
0, 0, 1, 1,	1, 1, 1, 1,	1, 1, 1, 1,	0, 1, 1, 1,
0, 0, 0, 0,	0, 0, 1, 1,	0, 0, 0, 0,	0, 0, 0, 0,
0, 0, 0, 0,	0, 1, 1, 1,	0, 0, 0, 1,	0, 0, 0, 0,
0, 0, 0, 1,	1, 1, 1, 1,	0, 1, 1, 1,	0, 0, 0, 1,
0, 0, 1, 1,	1, 1, 1, 1,	1, 1, 1, 1,	0, 1, 1, 1,
0, 0, 0, 0,	0, 0, 0, 0,	0, 0, 0, 0,	0, 0, 0, 0,
0, 1, 1, 1,	0, 0, 0, 0,	1, 1, 1, 1,	0, 0, 0, 0,
1, 1, 1, 1,	1, 1, 1, 1,	1, 1, 1, 1,	0, 0, 0, 0,
1, 1, 1, 1,	1, 1, 1, 1,	1, 1, 1, 1,	1, 1, 1, 1,
0, 0, 0, 0,	0, 1, 1, 1,	0, 0, 0, 0,	0, 1, 1, 1,
1, 0, 0, 0,	0, 0, 0, 1,	0, 0, 0, 0,	0, 0, 1, 1,
1, 1, 1, 0,	0, 0, 0, 0,	1, 0, 0, 0,	0, 0, 0, 1,
1, 1, 1, 1,	0, 0, 0, 0,	1, 1, 0, 0,	0, 0, 0, 0,
0, 0, 1, 1,	0, 0, 0, 0,	0, 0, 0, 0,	0, 1, 1, 1,
0, 0, 0, 1,	1, 0, 0, 0,	0, 0, 0, 0,	0, 0, 1, 1,
0, 0, 0, 0,	1, 1, 0, 0,	1, 0, 0, 0,	0, 0, 1, 1,
0, 0, 0, 0,	1, 1, 1, 0,	1, 1, 0, 0,	0, 0, 0, 1,
0, 0, 1, 1,	0, 0, 0, 0,	0, 1, 1, 0,	0, 0, 1, 1,
0, 0, 0, 1,	1, 0, 0, 0,	0, 1, 1, 0,	0, 1, 1, 0,
0, 0, 0, 1,	1, 0, 0, 0,	0, 1, 1, 0,	0, 1, 1, 0,
0, 0, 0, 0,	1, 1, 0, 0,	0, 1, 1, 0,	1, 1, 0, 0,
0, 0, 0, 1,	0, 0, 0, 0,	0, 1, 1, 1,	0, 0, 1, 1,
0, 1, 1, 1,	1, 1, 1, 1,	0, 0, 0, 1,	1, 0, 0, 1,
1, 1, 1, 0,	1, 1, 1, 1,	1, 0, 0, 0,	1, 0, 0, 1,
1, 0, 0, 0,	0, 0, 0, 0,	1, 1, 1, 0,	1, 1, 0, 0,

In this table of partition sets, the bolded and underlined entry is the location of the fix-up index for subset 1 (which is specified with one less bit). The fix-up index for subset 0 is always index 0, as the partitioning is always arranged such that index 0 is always in subset 0. Partition order goes from top-left to bottom-right, moving left to right and then top to bottom.

BC6H Compressed Endpoint Format

Header Bit	10 5 5 5	7 6 6 6	11 5 4 4	11 4 5 4	11 4 4 5	9 5 5 5	8 6 5 5	8 5 6 5	8 5 5 6	6 6 6 6	10 10	11 9	12 8	16 4
0														
1	m[1.0]	m[1.0]												
2	gy[4]	gy[5]												
3	by[4]	gz[4]												
4	bz[4]	gz[5]	m[4.0]	m[4.0]	m[4.0]	m[4.0]	m[4.0]	m[4.0]	m[4.0]	m[4.0]	m[4.0]	m[4.0]	m[4.0]	m[4.0]
5														
6														
7														
8														
9														
10														
11		rw[6:0]												
12		bz[0]												
13		bz[1]												
14		rw[9:0]	rw[9:0]	rw[9:0]	by[4]	rw[8:0]	gz[4]	rw[7:0]	bz[0]	rw[7:0]	bz[1]	bz[0]	bz[1]	rw[9:0]
15							by[4]	rw[9:0]						
16														
17														
18														
19														
20														
21		gw[6:0]												
22		by[5]												
23		bz[2]												
24	gw[9:0]	gy[4]	gw[9:0]	gw[9:0]	gw[9:0]	gy[4]	gw[8:0]	bz[2]	gw[7:0]	gy[5]	by[5]	gw[7:0]	gy[4]	gw[9:0]
25								gy[4]	gy[4]	gy[4]	gy[4]	gy[4]	gy[4]	gw[9:0]
26														
27														
28														
29														
30														
31		bw[6:0]												
32		bz[3]												
33		bz[5]												
34	bw[9:0]	bz[4]	bw[9:0]	bw[9:0]	bw[9:0]	bz[4]	bw[8:0]	bz[3]	bw[7:0]	bz[5]	bz[5]	bz[3]	bz[2]	bw[9:0]
35								bz[4]	bz[4]	bz[4]	bz[4]	bz[4]	bz[4]	bw[9:0]
36														
37														
38														
39	rx[4:0]	rx[5:0]	rx[4:0]	rw[10]	rx[3:0]	rw[10]	rx[3:0]	rx[4:0]	rx[5:0]	rx[4:0]	rx[4:0]	rx[5:0]	rx[3:0]	
40	gz[4]		gz[4]	gy[4]		gy[4]		gz[4]		gz[4]		rx[5:0]		
41														
42														
43														
44	gy[3:0]	gy[3:0]	gy[3:0]	gy[3:0]	gy[3:0]	gy[3:0]	gy[3:0]	gy[3:0]	gy[3:0]	gy[3:0]	gy[3:0]	rx[7:0]	rx[8:0]	rw[10:11]
45													rw[10:15]	
46														
47														
48														
49	gx[4:0]	gx[5:0]	gx[3:0]	gw[10]	gx[4:0]	gx[3:0]	gw[10]	bx[3:0]	gx[4:0]	gx[4:0]	gx[4:0]	gx[5:0]	gx[3:0]	
50	bz[0]	bz[0]	bz[0]	bz[0]	bz[0]	bz[0]	bz[0]	bx[4:0]	bz[0]	bx[4:0]	bz[0]	bx[5:0]	bx[5:0]	
51														
52														
53														
54	gz[3:0]	gz[3:0]	gz[3:0]	gz[3:0]	gz[3:0]	gz[3:0]	gz[3:0]	gz[3:0]	gz[3:0]	gz[3:0]	gz[3:0]	gz[9:0]	gx[8:0]	gw[10:11]
55													gw[10:15]	
56														
57														
58														
59	bx[4:0]	bx[5:0]	bx[3:0]	bw[10]	bx[3:0]	bx[4:0]	bw[10]	bx[4:0]	bx[4:0]	bx[4:0]	bx[4:0]	bx[5:0]	bx[5:0]	bx[3:0]
60	bz[1]	bz[1]	bz[1]	bz[1]	bz[1]	bz[1]	bz[1]	bz[1]	bz[1]	bz[1]	bz[1]	bz[1]	bz[1]	
61														
62														
63														
64	by[3:0]	by[3:0]	by[3:0]	by[3:0]	by[3:0]	by[3:0]	by[3:0]	by[3:0]	by[3:0]	by[3:0]	by[3:0]	bx[8:0]	bx[8:0]	bw[10:11]
65													bw[10:15]	
66														
67														
68														
69	ry[4:0]	ry[5:0]	ry[4:0]	bz[2]	rz[3:0]	rz[3:0]	bz[2]	rz[4:0]	ry[4:0]	rz[4:0]	rz[4:0]	ry[5:0]		
70	bz[2]	bz[2]	bz[2]	bz[2]	bz[2]	bz[2]	bz[2]	bz[2]	ry[5:0]	bz[2]	bz[2]	ry[5:0]		
71														
72														
73														
74														
75	rz[4:0]	rz[5:0]	rz[4:0]	rz[4:0]	gy[4]	rz[3:0]	rz[3:0]	rz[4:0]	rz[5:0]	rz[4:0]	rz[4:0]	rz[5:0]		
76	bz[3]	bz[3]	bz[3]	bz[3]	bz[3]	bz[3]	bz[3]	bz[3]	bz[3]	bz[3]	bz[3]	bz[3]		
77														
78														
79														
80														
81	d[4:0]	d[4:0]	d[4:0]	d[4:0]	d[4:0]	d[4:0]	d[4:0]	d[4:0]	d[4:0]	d[4:0]	d[4:0]	d[4:0]	d[4:0]	16 4
	10 5 5 5	7 6 6 6	11 5 4 4	11 4 5 4	11 4 4 5	9 5 5 5	8 6 5 5	8 5 6 5	8 5 5 6	6 6 6 6	10 10	11 9	12 8	16 4

This table shows the bit fields for the compressed endpoints as a function of the endpoint format, with each column specifying an encoding and each row specifying a bit field. This approach takes up 82 bits for two-region tiles and 65 bits for one-region tiles. As an example, the first 5 bits for the one-region [16 4] encoding above (specifically the right-most column) are bits m[4:0], the next 10 bits are bits rw[9:0], and so on with the last 6 bits containing bw[10:15].

The field names in the table above are defined as follows:

FIELD	VARIABLE
m	mode
d	shape index
rw	endpt[0].A[0]
rx	endpt[0].B[0]
ry	endpt[1].A[0]
rz	endpt[1].B[0]
gw	endpt[0].A[1]
gx	endpt[0].B[1]
gy	endpt[1].A[1]
gz	endpt[1].B[1]
bw	endpt[0].A[2]
bx	endpt[0].B[2]
by	endpt[1].A[2]
bz	endpt[1].B[2]

Endpt[i], where i is either 0 or 1, refers to the 0th or 1st set of endpoints respectively.

Sign Extension for Endpoint Values

For two-region tiles, there are four endpoint values that can be sign extended. Endpt[0].A is signed only if the format is a signed format; the other endpoints are signed only if the endpoint was transformed, or if the format is a signed format. The code below demonstrates the algorithm for extending the sign of two-region endpoint values.

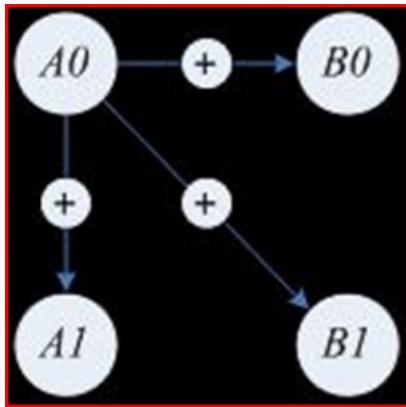
```
static void sign_extend_two_region(Pattern &p, IntEndpts endpts[NREGIONS_TWO])
{
    for (int i=0; i<NCHANNELS; ++i)
    {
        if (BC6H::FORMAT == SIGNED_F16)
            endpts[0].A[i] = SIGN_EXTEND(endpts[0].A[i], p.chan[i].prec);
        if (p.transformed || BC6H::FORMAT == SIGNED_F16)
        {
            endpts[0].B[i] = SIGN_EXTEND(endpts[0].B[i], p.chan[i].delta[0]);
            endpts[1].A[i] = SIGN_EXTEND(endpts[1].A[i], p.chan[i].delta[1]);
            endpts[1].B[i] = SIGN_EXTEND(endpts[1].B[i], p.chan[i].delta[2]);
        }
    }
}
```

For one-region tiles, the behavior is the same, only with endpt[1] removed.

```
static void sign_extend_one_region(Pattern &p, IntEndpts endpts[NREGIONS_ONE])
{
    for (int i=0; i<NCHANNELS; ++i)
    {
        if (BC6H::FORMAT == SIGNED_F16)
            endpts[0].A[i] = SIGN_EXTEND(endpts[0].A[i], p.chan[i].prec);
        if (p.transformed || BC6H::FORMAT == SIGNED_F16)
            endpts[0].B[i] = SIGN_EXTEND(endpts[0].B[i], p.chan[i].delta[0]);
    }
}
```

Transform Inversion for Endpoint Values

For two-region tiles, the transform applies the inverse of the difference encoding, adding the base value at endpt[0].A to the three other entries for a total of 9 add operations. In the image below, the base value is represented as "A0" and has the highest floating point precision. "A1," "B0," and "B1" are all deltas calculated from the anchor value, and these delta values are represented with lower precision. (A0 corresponds to endpt[0].A, B0 corresponds to endpt[0].B, A1 corresponds to endpt[1].A, and B1 corresponds to endpt[1].B.)



For one-region tiles there is only one delta offset, and therefore only 3 add operations.

The decompressor must ensure that the results of the inverse transform will not overflow the precision of endpt[0].a. In the case of an overflow, the values resulting from the inverse transform must wrap within the same number of bits. If the precision of A0 is "p" bits, then the transform algorithm is:

$$B0 = (B0 + A0) \& ((1 \ll p) - 1)$$

For signed formats, the results of the delta calculation must be sign extended as well. If the sign extension operation considers extending both signs, where 0 is positive and 1 is negative, then the sign extension of 0 takes care of the clamp above. Equivalently, after the clamp above, only a value of 1 (negative) needs to be sign extended.

Unquantization of Color Endpoints

Given the uncompressed endpoints, the next step is to perform an initial unquantization of the color endpoints. This involves three steps:

- An unquantization of the color palettes
- Interpolation of the palettes
- Unquantization finalization

Separating the unquantization process into two parts (color palette unquantization before interpolation and final unquantization after interpolation) reduces the number of multiplication operations required when

compared to a full unquantization process before palette interpolation.

The code below illustrates the process for retrieving estimates of the original 16-bit color values, and then using the supplied weight values to add 6 additional color values to the palette. The same operation is performed on each channel.

```
int aWeight3[] = {0, 9, 18, 27, 37, 46, 55, 64};
int aWeight4[] = {0, 4, 9, 13, 17, 21, 26, 30, 34, 38, 43, 47, 51, 55, 60, 64};

// c1, c2: endpoints of a component
void generate_palette_unquantized(UINT8 uNumIndices, int c1, int c2, int prec, UINT16 palette[NINDICES])
{
    int* aWeights;
    if(uNumIndices == 8)
        aWeights = aWeight3;
    else // uNumIndices == 16
        aWeights = aWeight4;

    int a = unquantize(c1, prec);
    int b = unquantize(c2, prec);

    // interpolate
    for(int i = 0; i < uNumIndices; ++i)
        palette[i] = finish_unquantize((a * (64 - aWeights[i]) + b * aWeights[i] + 32) >> 6);
}
```

The next code sample demonstrates the interpolation process, with the following observations:

- Since the full range of color values for the **unquantize** function (below) are from -32768 to 65535, the interpolator is implemented using 17-bit signed arithmetic.
- After interpolation, the values are passed to the **finish_unquantize** function (described in the third sample in this section), which applies the final scaling.
- All hardware decompressors are required to return bit-accurate results with these functions.

```

int unquantize(int comp, int uBitsPerComp)
{
    int unq, s = 0;
    switch(BC6H::FORMAT)
    {
        case UNSIGNED_F16:
            if(uBitsPerComp >= 15)
                unq = comp;
            else if(comp == 0)
                unq = 0;
            else if(comp == ((1 << uBitsPerComp) - 1))
                unq = 0xFFFF;
            else
                unq = ((comp << 16) + 0x8000) >> uBitsPerComp;
            break;

        case SIGNED_F16:
            if(uBitsPerComp >= 16)
                unq = comp;
            else
            {
                if(comp < 0)
                {
                    s = 1;
                    comp = -comp;
                }

                if(comp == 0)
                    unq = 0;
                else if(comp >= ((1 << (uBitsPerComp - 1)) - 1))
                    unq = 0x7FFF;
                else
                    unq = ((comp << 15) + 0x4000) >> (uBitsPerComp-1);

                if(s)
                    unq = -unq;
            }
            break;
    }
    return unq;
}

```

finish_unquantize is called after palette interpolation. The **unquantize** function postpones the scaling by 31/32 for signed, 31/64 for unsigned. This behavior is required to get the final value into valid half range(-0x7BFF ~ 0x7BFF) after the palette interpolation is completed in order to reduce the number of necessary multiplications. **finish_unquantize** applies the final scaling and returns an **unsigned short** value that gets reinterpreted into **half**.

```
unsigned short finish_unquantize(int comp)
{
    if(BC6H::FORMAT == UNSIGNED_F16)
    {
        comp = (comp * 31) >> 6;                                // scale the magnitude by 31/64
        return (unsigned short) comp;
    }
    else // (BC6H::FORMAT == SIGNED_F16)
    {
        comp = (comp < 0) ? -((-comp) * 31) >> 5 : (comp * 31) >> 5; // scale the magnitude by 31/32
        int s = 0;
        if(comp < 0)
        {
            s = 0x8000;
            comp = -comp;
        }
        return (unsigned short) (s | comp);
    }
}
```

Related topics

[Texture Block Compression in Direct3D 11](#)

BC7 Format

11/2/2020 • 7 minutes to read • [Edit Online](#)

The BC7 format is a texture compression format used for high-quality compression of RGB and RGBA data.

- [About BC7/DXGI_FORMAT_BC7](#)
- [BC7 Implementation](#)
- [Decoding the BC7 Format](#)
- [Related topics](#)

For info about the block modes of the BC7 format, see [BC7 Format Mode Reference](#).

About BC7/DXGI_FORMAT_BC7

BC7 is specified by the following DXGI_FORMAT enumeration values:

- `DXGI_FORMAT_BC7_TYPELESS`.
- `DXGI_FORMAT_BC7_UNORM`.
- `DXGI_FORMAT_BC7_UNORM_SRGB`.

The BC7 format can be used for [Texture2D](#) (including arrays), [Texture3D](#), or [TextureCube](#) (including arrays) texture resources. Similarly, this format applies to any MIP-map surfaces associated with these resources.

BC7 uses a fixed block size of 16 bytes (128 bits) and a fixed tile size of 4x4 texels. As with previous BC formats, texture images larger than the supported tile size (4x4) are compressed by using multiple blocks. This addressing identity also applies to three-dimensional images and MIP-maps, cubemaps, and texture arrays. All image tiles must be of the same format.

BC7 compresses both three-channel (RGB) and four-channel (RGBA) fixed-point data images. Typically, source data is 8 bits per color component (channel), although the format is capable of encoding source data with higher bits per color component. All image tiles must be of the same format.

The BC7 decoder performs decompression before texture filtering is applied.

BC7 decompression hardware must be bit accurate; that is, the hardware must return results that are identical to the results returned by the decoder described in this document.

BC7 Implementation

A BC7 implementation can specify one of 8 modes, with the mode specified in the least significant bit of the 16 byte (128 bit) block. The mode is encoded by zero or more bits with a value of 0 followed by a 1.

A BC7 block can contain multiple endpoint pairs. For the purposes of this documentation, the set of indices that correspond to an endpoint pair may be referred to as a "subset." Also, in some block modes, the endpoint representation is encoded in a form that -- again, for the purposes of this documentation -- shall be referred to as "RBGP" where the "P" bit represents a shared least significant bit for the color components of the endpoint. For example, if the endpoint representation for the format is "RGB 5.5.5.1," then the endpoint is interpreted as an RGB 6.6.6 value, where the state of the P-bit defines the least significant bit of each component. Similarly, for source data with an alpha channel, if the representation for the format is "RGBAP 5.5.5.5.1," then the endpoint is interpreted as RGBA 6.6.6.6. Depending on the block mode, you can specify the shared least significant bit for either both endpoints of a subset individually (2 P-bits per subset), or shared between endpoints of a subset (1 P-bit per subset).

For BC7 blocks that don't explicitly encode the alpha component, a BC7 block consists of mode bits, partition bits, compressed endpoints, compressed indices, and an optional P-bit. In these blocks the endpoints have an RGB-only representation and the alpha component is decoded as 1.0 for all texels in the source data.

For BC7 blocks that have combined color and alpha components, a block consists of mode bits, compressed endpoints, compressed indices, and optional partition bits and a P-bit. In these blocks, the endpoint colors are expressed in RGBA format, and alpha component values are interpolated alongside the color component values.

For BC7 blocks that have separate color and alpha components, a block consists of mode bits, rotation bits, compressed endpoints, compressed indices, and an optional index selector bit. These blocks have an effective RGB vector [R, G, B] and a scalar alpha channel [A] separately encoded.

The following table lists the components of each block type.

BC7 BLOCK CONTAINS..	MODE BITS	ROTATION BITS	INDEX SELECTOR BIT	PARTITION BITS	COMPRESSED ENDPOINTS	P-BIT	COMPRESSED INDICES
color components only	required	N/A	N/A	required	required	optional	required
color + alpha combined	required	N/A	N/A	optional	required	optional	required
color and alpha separated	required	required	optional	N/A	required	N/A	required

BC7 defines a palette of colors on an approximate line between two endpoints. The mode value determines the number of interpolating endpoint pairs per block. BC7 stores one palette index per texel.

For each subset of indices that corresponds to a pair of endpoints, the encoder fixes the state of one bit of the compressed index data for that subset. It does so by choosing an endpoint order that allows the index for the designated "fix-up" index to set its most significant bit to 0, and which can then be discarded, saving one bit per subset. For block modes with only a single subset, the fix-up index is always index 0.

Decoding the BC7 Format

The following pseudocode outlines the steps to decompress the pixel at (x,y) given the 16 byte BC7 block.

```

decompress_bc7(x, y, block)
{
    mode = extract_mode(block);

    //decode partition data from explicit partition bits
    subset_index = 0;
    num_subsets = 1;

    if (mode.type == 0 OR == 1 OR == 2 OR == 3 OR == 7)
    {
        num_subsets = get_num_subsets(mode.type);
        partition_set_id = extract_partition_set_id(mode, block);
        subset_index = get_partition_index(num_subsets, partition_set_id, x, y);
    }

    //extract raw, compressed endpoint bits
    UINT8 endpoint_array[num_subsets][4] = extract_endpoints(mode, block);

    //decode endpoint color and alpha for each subset
    fully_decode_endpoints(endpoint_array, mode, block);

    //endpoints are now complete.
    UINT8 endpoint_start[4] = endpoint_array[2 * subset_index];
    UINT8 endpoint_end[4] = endpoint_array[2 * subset_index + 1];

    //Determine the palette index for this pixel
    alpha_index = get_alpha_index(block, mode, x, y);
    alpha_bitcount = get_alpha_bitcount(block, mode);
    color_index = get_color_index(block, mode, x, y);
    color_bitcount = get_color_bitcount(block, mode);

    //determine output
    UINT8 output[4];
    output.rgb = interpolate(endpoint_start.rgb, endpoint_end.rgb, color_index, color_bitcount);
    output.a = interpolate(endpoint_start.a, endpoint_end.a, alpha_index, alpha_bitcount);

    if (mode.type == 4 OR == 5)
    {
        //Decode the 2 color rotation bits as follows:
        // 00 - Block format is Scalar(A) Vector(RGB) - no swapping
        // 01 - Block format is Scalar(R) Vector(AGB) - swap A and R
        // 10 - Block format is Scalar(G) Vector(RAB) - swap A and G
        // 11 - Block format is Scalar(B) Vector(RGA) - swap A and B
        rotation = extract_rot_bits(mode, block);
        output = swap_channels(output, rotation);
    }
}

```

The following pseudocode outlines the steps to fully decode endpoint color and alpha components for each subset given a 16-byte BC7 block.

```

fully_decode_endpoints(endpoint_array, mode, block)
{
    //first handle modes that have P-bits
    if (mode.type == 0 OR == 1 OR == 3 OR == 6 OR == 7)
    {
        for each endpoint i
        {
            //component-wise left-shift
            endpoint_array[i].rgba = endpoint_array[i].rgba << 1;
        }

        //if P-bit is shared
        if (mode.type == 1)
        {
            pbit_zero = extract_pbit_zero(mode, block);
            pbit_one = extract_pbit_one(mode, block);

            //rgb component-wise insert pbits
            endpoint_array[0].rgb |= pbit_zero;
            endpoint_array[1].rgb |= pbit_zero;
            endpoint_array[2].rgb |= pbit_one;
            endpoint_array[3].rgb |= pbit_one;
        }
        else //unique P-bit per endpoint
        {
            pbit_array = extract_pbit_array(mode, block);
            for each endpoint i
            {
                endpoint_array[i].rgba |= pbit_array[i];
            }
        }
    }

    for each endpoint i
    {
        // Color_component_precision & alpha_component_precision includes pbit
        // left shift endpoint components so that their MSB lies in bit 7
        endpoint_array[i].rgb = endpoint_array[i].rgb << (8 - color_component_precision(mode));
        endpoint_array[i].a = endpoint_array[i].a << (8 - alpha_component_precision(mode));

        // Replicate each component's MSB into the LSBs revealed by the left-shift operation above
        endpoint_array[i].rgb = endpoint_array[i].rgb | (endpoint_array[i].rgb >>
color_component_precision(mode));
        endpoint_array[i].a = endpoint_array[i].a | (endpoint_array[i].a >>
alpha_component_precision(mode));
    }

    //If this mode does not explicitly define the alpha component
    //set alpha equal to 1.0
    if (mode.type == 0 OR == 1 OR == 2 OR == 3)
    {
        for each endpoint i
        {
            endpoint_array[i].a = 255; //i.e. alpha = 1.0f
        }
    }
}

```

To generate each interpolated component for each subset, use the following algorithm: let "c" be the component to generate; let "e0" be that component of endpoint 0 of the subset; and let "e1" be that component of endpoint 1 of the subset.

```

UINT16 aWeight2[] = {0, 21, 43, 64};
UINT16 aWeight3[] = {0, 9, 18, 27, 37, 46, 55, 64};
UINT16 aWeight4[] = {0, 4, 9, 13, 17, 21, 26, 30, 34, 38, 43, 47, 51, 55, 60, 64};

UINT8 interpolate(UINT8 e0, UINT8 e1, UINT8 index, UINT8 indexprecision)
{
    if(indexprecision == 2)
        return (UINT8) (((64 - aWeights2[index])*UINT16(e0) + aWeights2[index]*UINT16(e1) + 32) >> 6);
    else if(indexprecision == 3)
        return (UINT8) (((64 - aWeights3[index])*UINT16(e0) + aWeights3[index]*UINT16(e1) + 32) >> 6);
    else // indexprecision == 4
        return (UINT8) (((64 - aWeights4[index])*UINT16(e0) + aWeights4[index]*UINT16(e1) + 32) >> 6);
}

```

The following pseudocode illustrates how to extract indices and bit counts for color and alpha components. Blocks with separate color and alpha also have two sets of index data: one for the vector channel, and one for the scalar channel. For Mode 4, these indices are of differing widths (2 or 3 bits), and there is a one-bit selector which specifies whether the vector or scalar data uses the 3-bit indices. (Extracting the alpha bit count is similar to extracting color bit count but with inverse behavior based on the **idxMode** bit.)

```

bitcount get_color_bitcount(block, mode)
{
    if (mode.type == 0 OR == 1)
        return 3;

    if (mode.type == 2 OR == 3 OR == 5 OR == 7)
        return 2;

    if (mode.type == 6)
        return 4;

    //The only remaining case is Mode 4 with 1-bit index selector
    idxMode = extract_idxMode(block);
    if (idxMode == 0)
        return 2;
    else
        return 3;
}

```

Related topics

[Texture Block Compression in Direct3D 11](#)

BC7 Format Mode Reference

2/22/2020 • 3 minutes to read • [Edit Online](#)

This documentation contains a list of the 8 block modes and bit allocations for BC7 texture compression format blocks.

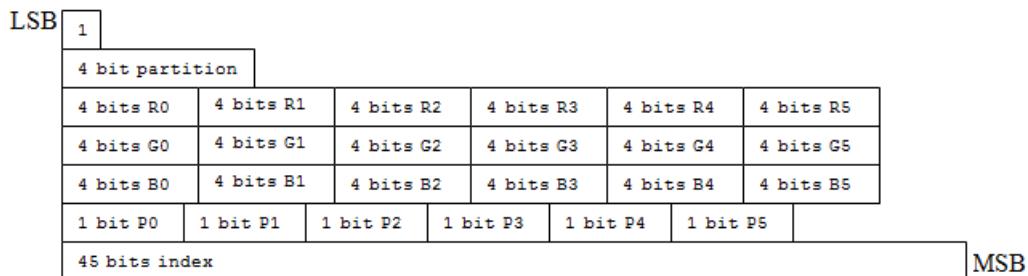
The colors for each subset within a block are represented by two explicit endpoint colors and a set of interpolated colors between them. Depending on the block's index precision, each subset can have 4, 8 or 16 possible colors.

- [Mode 0](#)
- [Mode 1](#)
- [Mode 2](#)
- [Mode 3](#)
- [Mode 4](#)
- [Mode 5](#)
- [Mode 6](#)
- [Mode 7](#)
- [Remarks](#)
- [Related topics](#)

Mode 0

BC7 Mode 0 has the following characteristics:

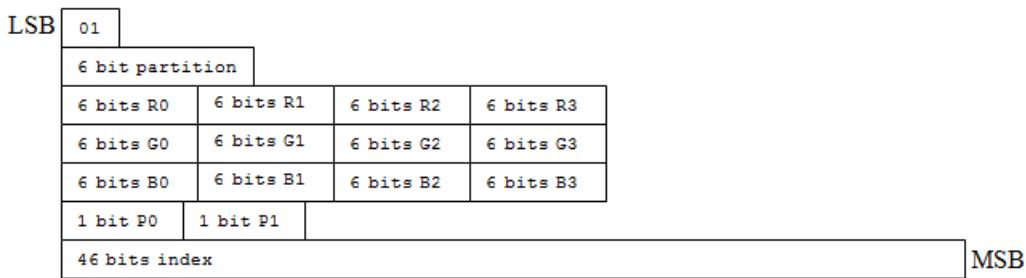
- Color components only (no alpha)
- 3 subsets per block
- RGBP 4.4.4.1 endpoints with a unique P-bit per endpoint
- 3-bit indices
- 16 partitions



Mode 1

BC7 Mode 1 has the following characteristics:

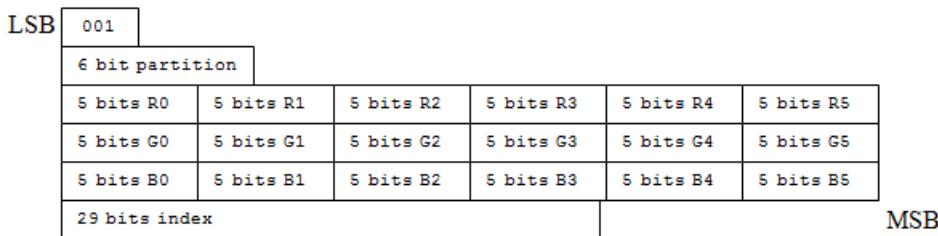
- Color components only (no alpha)
- 2 subsets per block
- RGBP 6.6.6.1 endpoints with a shared P-bit per subset)
- 3-bit indices
- 64 partitions



Mode 2

BC7 Mode 2 has the following characteristics:

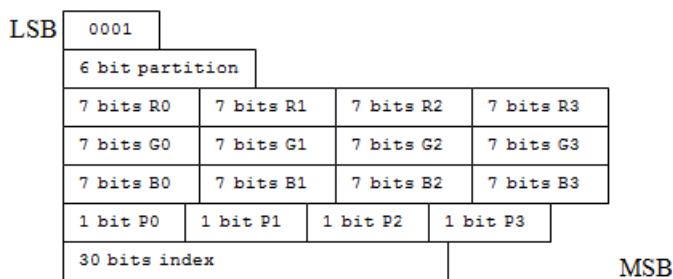
- Color components only (no alpha)
- 3 subsets per block
- RGB 5.5.5 endpoints
- 2-bit indices
- 64 partitions



Mode 3

BC7 Mode 3 has the following characteristics:

- Color components only (no alpha)
- 2 subsets per block
- RGBP 7.7.7.1 endpoints with a unique P-bit per subset)
- 2-bit indices
- 64 partitions

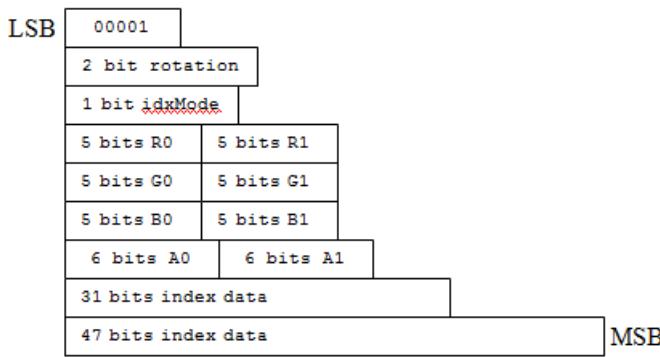


Mode 4

BC7 Mode 4 has the following characteristics:

- Color components with separate alpha component
- 1 subset per block
- RGB 5.5.5 color endpoints
- 6-bit alpha endpoints
- 16 x 2-bit indices

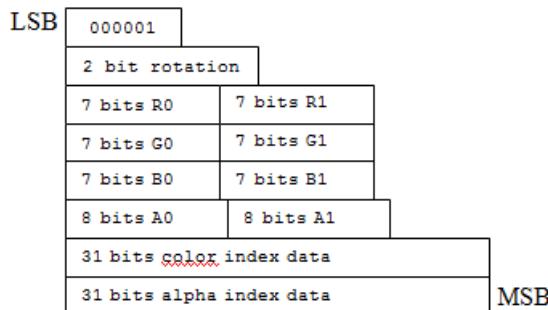
- 16 x 3-bit indices
- 2-bit component rotation
- 1-bit index selector (whether the 2- or 3-bit indices are used)



Mode 5

BC7 Mode 5 has the following characteristics:

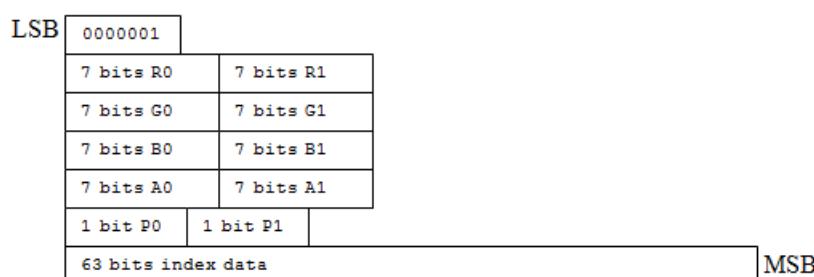
- Color components with separate alpha component
- 1 subset per block
- RGB 7.7.7 color endpoints
- 8-bit alpha endpoints
- 16 x 2-bit color indices
- 16 x 2-bit alpha indices
- 2-bit component rotation



Mode 6

BC7 Mode 6 has the following characteristics:

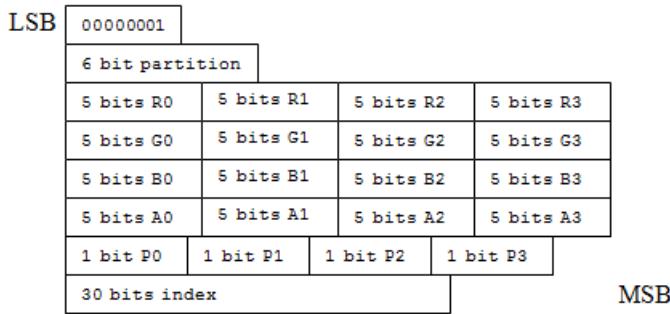
- Combined color and alpha components
- One subset per block
- RGBAP 7.7.7.7.1 color (and alpha) endpoints (unique P-bit per endpoint)
- 16 x 4-bit indices



Mode 7

BC7 Mode 7 has the following characteristics:

- Combined color and alpha components
- 2 subsets per block
- RGBAP 5.5.5.5.1 color (and alpha) endpoints (unique P-bit per endpoint)
- 2-bit indices
- 64 partitions



Remarks

Mode 8 (the least significant byte is set to 0x00) is reserved. Don't use it in your encoder. If you pass this mode to the hardware, a block initialized to all zeroes is returned.

In BC7, you can encode the alpha component in one of the following ways:

- Block types without explicit alpha component encoding. In these blocks, the color endpoints have an RGB-only encoding, with the alpha component decoded to 1.0 for all texels.
- Block types with combined color and alpha components. In these blocks, the endpoint color values are specified in the RGBA format, and the alpha component values are interpolated along with the color values.
- Block types with separated color and alpha components. In these blocks the color and alpha values are specified separately, each with their own set of indices. As a result, they have an effective vector and a scalar channel separately encoded, where the vector commonly specifies the color channels [R, G, B] and the scalar specifies the alpha channel [A]. To support this approach, a separate 2-bit field is provided in the encoding, which permits the specification of the separate channel encoding as a scalar value. As a result, the block can have one of the following four different representations of this alpha encoding (as indicated by the 2-bit field):
 - RGB|A: alpha channel separate
 - AGB|R: "red" color channel separate
 - RAB|G: "green" color channel separate
 - RGA|B: "blue" color channel separate

The decoder reorders the channel order back to RGBA after decoding, so the internal block format is invisible to the developer. Blacks with separate color and alpha components also have two sets of index data: one for the vectored set of channels, and one for the scalar channel. (In the case of Mode 4, these indices are of differing widths [2 or 3 bits]. Mode 4 also contains a 1-bit selector that specifies whether the vector or the scalar channel uses the 3-bit indices.)

Related topics

[BC7 Format](#)

Floating-point rules (Direct3D 11)

11/2/2020 • 6 minutes to read • [Edit Online](#)

Direct3D 11 supports several floating-point representations. All floating-point computations operate under a defined subset of the IEEE 754 32-bit single precision floating-point rules.

- [32-bit floating-point rules](#)
 - Honored IEEE-754 rules
 - Deviations or additional requirements from IEEE-754 rules
- [64-bit \(double precision\) floating point rules](#)
- [16-bit floating-point rules](#)
- [11-bit and 10-bit floating-point rules](#)
- [Related topics](#)

32-bit floating-point rules

There are two sets of rules: those that conform to IEEE-754, and those that deviate from the standard.

Honored IEEE-754 rules

Some of these rules are a single option where IEEE-754 offers choices.

- Divide by 0 produces +/- INF, except 0/0 which results in NaN.
- log of (+/-) 0 produces -INF. log of a negative value (other than -0) produces NaN.
- Reciprocal square root (rsq) or square root (sqrt) of a negative number produces NaN. The exception is -0; sqrt(-0) produces -0, and rsq(-0) produces -INF.
- INF - INF = NaN
- (+/-)INF / (+/-)INF = NaN
- (+/-)INF * 0 = NaN
- NaN (any OP) any-value = NaN
- The comparisons EQ, GT, GE, LT, and LE, when either or both operands is NaN returns FALSE.
- Comparisons ignore the sign of 0 (so +0 equals -0).
- The comparison NE, when either or both operands is NaN returns TRUE.
- Comparisons of any non-NaN value against +/- INF return the correct result.

Deviations or additional requirements from IEEE-754 rules

- IEEE-754 requires floating-point operations to produce a result that is the nearest representable value to an infinitely-precise result, known as round-to-nearest-even. Direct3D 11 defines the same requirement: 32-bit floating-point operations produce a result that is within 0.5 unit-last-place (ULP) of the infinitely-precise result. This means that, for example, hardware is allowed to truncate results to 32-bit rather than perform round-to-nearest-even, as that would result in error of at most 0.5 ULP. This rule applies only to addition, subtraction, and multiplication.
- There is no support for floating-point exceptions, status bits or traps.
- Denorms are flushed to sign-preserved zero on input and output of any floating-point mathematical operation. Exceptions are made for any I/O or data movement operation that doesn't manipulate the data.
- States that contain floating-point values, such as Viewport MinDepth/MaxDepth, BorderColor values, may be provided as denorm values and may or may not be flushed before the hardware uses them.

- Min or max operations flush denorms for comparison, but the result may or may not be denorm flushed.
- NaN input to an operation always produces NaN on output. But the exact bit pattern of the NaN is not required to stay the same (unless the operation is a raw move instruction - which doesn't alter data.)
- Min or max operations for which only one operand is NaN return the other operand as the result (contrary to comparison rules we looked at earlier). This is a IEEE 754R rule.

The IEEE-754R specification for floating point min and max operations states that if one of the inputs to min or max is a quiet QNaN value, the result of the operation is the other parameter. For example:

```
min(x,QNaN) == min(QNaN,x) == x (same for max)
```

A revision of the IEEE-754R specification adopted a different behavior for min and max when one input is a "signaling" SNaN value versus a QNaN value:

```
min(x,SNaN) == min(SNaN,x) == QNaN (same for max)
```

Generally, Direct3D follows the standards for arithmetic: IEEE-754 and IEEE-754R. But in this case, we have a deviation.

The arithmetic rules in Direct3D 10 and later don't make any distinctions between quiet and signaling NaN values (QNaN versus SNaN). All NaN values are handled the same way. In the case of min and max, the Direct3D behavior for any NaN value is like how QNaN is handled in the IEEE-754R definition. (For completeness - if both inputs are NaN, any NaN value is returned.)

- Another IEEE 754R rule is that $\min(-0,+0) == \min(+0,-0) == -0$, and $\max(-0,+0) == \max(+0,-0) == +0$, which honors the sign, in contrast to the comparison rules for signed zero (as we saw earlier). Direct3D recommends the IEEE 754R behavior here, but doesn't enforce it; it is permissible for the result of comparing zeros to be dependent on the order of parameters, using a comparison that ignores the signs.
- $x * 1.0f$ always results in x (except denorm flushed).
- $x / 1.0f$ always results in x (except denorm flushed).
- $x +/- 0.0f$ always results in x (except denorm flushed). But $-0 + 0 = +0$.
- Fused operations (such as mad, dp3) produce results that are no less accurate than the worst possible serial ordering of evaluation of the unfused expansion of the operation. The definition of the worst possible ordering, for the purpose of tolerance, is not a fixed definition for a given fused operation; it depends on the particular values of the inputs. The individual steps in the unfused expansion are each allowed 1 ULP tolerance (or for any instructions Direct3D calls out with a more lax tolerance than 1 ULP, the more lax tolerance is allowed).
- Fused operations adhere to the same NaN rules as non-fused operations.
- sqrt and rcp have 1 ULP tolerance. The shader reciprocal and reciprocal square-root instructions, [rcp](#) and [rsq](#), have their own separate relaxed precision requirement.
- Multiply and divide each operate at the 32-bit floating-point precision level (accuracy to 0.5 ULP for multiply, 1.0 ULP for reciprocal). If x/y is implemented directly, results must be of greater or equal accuracy than a two-step method.

64-bit (double precision) floating point rules

Hardware and display drivers optionally support double-precision floating-point. To indicate support, when you

call `ID3D11Device::CheckFeatureSupport` with `D3D11_FEATURE_DOUBLE_PRECISION`, the driver sets `DoublePrecisionFloatShaderOps` of `D3D11_FEATURE_DATA_DOUBLE_PRECISION` to TRUE. The driver and hardware must then support all double-precision floating-point instructions.

Double-precision instructions follow IEEE 754R behavior requirements.

Support for generation of denormalized values is required for double-precision data (no flush-to-zero behavior). Likewise, instructions don't read denormalized data as a signed zero, they honor the denorm value.

16-bit floating-point rules

Direct3D 11 also supports 16-bit representations of floating-point numbers.

Format:

- 1 sign bit (s) in the MSB bit position
- 5 bits of biased exponent (e)
- 10 bits of fraction (f), with an additional hidden bit

A float16 value (v) follows these rules:

- if $e == 31$ and $f != 0$, then v is NaN regardless of s
- if $e == 31$ and $f == 0$, then $v = (-1)s * \infty$ (signed infinity)
- if e is between 0 and 31, then $v = (-1)s * 2^{(e-15)} * (1.f)$
- if $e == 0$ and $f != 0$, then $v = (-1)s * 2^{(e-14)} * (0.f)$ (denormalized numbers)
- if $e == 0$ and $f == 0$, then $v = (-1)s * 0$ (signed zero)

32-bit floating-point rules also hold for 16-bit floating-point numbers, adjusted for the bit layout described earlier. Exceptions to this include:

- Precision: Unfused operations on 16-bit floating-point numbers produce a result that is the nearest representable value to an infinitely-precise result (round to nearest even, per IEEE-754, applied to 16-bit values). 32-bit floating-point rules adhere to 1 ULP tolerance, 16-bit floating-point rules adhere to 0.5 ULP for unfused operations, and 0.6 ULP for fused operations.
- 16-bit floating-point numbers preserve denorms.

11-bit and 10-bit floating-point rules

Direct3D 11 also supports 11-bit and 10-bit floating-point formats.

Format:

- No sign bit
- 5 bits of biased exponent (e)
- 6 bits of fraction (f) for an 11-bit format, 5 bits of fraction (f) for a 10-bit format, with an additional hidden bit in either case.

A float11/float10 value (v) follows the following rules:

- if $e == 31$ and $f != 0$, then v is NaN
- if $e == 31$ and $f == 0$, then $v = +\infty$
- if e is between 0 and 31, then $v = 2^{(e-15)} * (1.f)$
- if $e == 0$ and $f != 0$, then $v = 2^{(e-14)} * (0.f)$ (denormalized numbers)
- if $e == 0$ and $f == 0$, then $v = 0$ (zero)

32-bit floating-point rules also hold for 11-bit and 10-bit floating-point numbers, adjusted for the bit layout

described earlier. Exceptions include:

- Precision: 32-bit floating-point rules adhere to 0.5 ULP.
- 10/11-bit floating-point numbers preserve denorms.
- Any operation that would result in a number less than zero is clamped to zero.

Related topics

[Resources](#)

[Textures](#)

Tiled resources

2/22/2020 • 2 minutes to read • [Edit Online](#)

Tiled resources can be thought of as large logical resources that use small amounts of physical memory.

This section describes why tiled resources are needed and how you create and use tiled resources.

In this section

TOPIC	DESCRIPTION
Why are tiled resources needed?	Tiled resources are needed so less graphics processing unit (GPU) memory is wasted storing regions of surfaces that the application knows will not be accessed, and the hardware can understand how to filter across adjacent tiles.
Creating tiled resources	Tiled resources are created by specifying the D3D11_RESOURCE_MISC_TILED flag when you create a resource.
Tiled Resource APIs	The APIs described in this section work with tiled resources and tile pool.
Pipeline access to tiled resources	Tiled resources can be used in shader resource views (SRV), render target views (RTV), depth stencil views (DSV) and unordered access views (UAV), as well as some bind points where views aren't used, such as vertex buffer bindings.
Tiled resources features tiers	Direct3D 11.2 exposes tiled resources support in two tiers with the D3D11_TILED_RESOURCES_TIER values.

Related topics

[Resources](#)

Why are tiled resources needed?

11/2/2020 • 5 minutes to read • [Edit Online](#)

Tiled resources are needed so less graphics processing unit (GPU) memory is wasted storing regions of surfaces that the application knows will not be accessed, and the hardware can understand how to filter across adjacent tiles.

In a graphics system (that is, the operating system, display driver, and graphics hardware) without tiled resource support, the graphics system manages all Direct3D memory allocations at subresource granularity. For a [Buffer](#), the entire Buffer is the subresource. For a [Texture](#) (for example, [Texture2D](#)), each mip level is a subresource; for a texture array (for example, [Texture2DArray](#)), each mip level at a given array slice is a subresource. The graphics system only exposes the ability to manage the mapping of allocations at this subresource granularity. In the context of tiled resources, "mapping" refers to making data visible to the GPU.

Suppose an application knows that a particular rendering operation only needs to access a small portion of an image mipmap chain (perhaps not even the full area of a given mipmap). Ideally, the app could inform the graphics system about this need. The graphics system would then only bother to ensure that the needed memory is mapped on the GPU without paging in too much memory. In reality, without tiled resource support, the graphics system can only be informed about the memory that needs to be mapped on the GPU at subresource granularity (for example, a range of full mipmap levels that could be accessed). There is no demand faulting in the graphics system either, so potentially a lot of excess GPU memory must be used to make full subresources mapped before a rendering command that references any part of the memory is executed. This is just one issue that makes the use of large memory allocations difficult in Direct3D without tiled resource support.

Direct3D 11 supports [Texture2D](#) surfaces with up to 16384 pixels on a given side. An image that is 16384 wide by 16384 tall and 4 bytes per pixel would consume 1GB of video memory (and adding mipmaps would double that amount). In practice, all 1GB would rarely need to be referenced in a single rendering operation.

Some game developers model terrain surfaces as large as 128K by 128K. The way they get this to work on existing GPUs is to break the surface into tiles that are small enough for hardware to handle. The application must figure out which tiles might be needed and load them into a cache of textures on the GPU - a software paging system. A significant downside to this approach comes from the hardware not knowing anything about the paging that is going on: When a part of an image needs to be shown on screen that straddles tiles, the hardware does not know how to perform fixed function (that is, efficient) filtering across tiles. This means the application managing its own software tiling must resort to manual texture filtering in shader code (which becomes very expensive if a good quality anisotropic filter is desired) and/or waste memory authoring gutters around tiles that contain data from neighboring tiles so that fixed function hardware filtering can continue to provide some assistance.

If a tiled representation of surface allocations could be a first class feature in the graphics system, the application could tell the hardware which tiles to make available. In this way, less GPU memory is wasted storing regions of surfaces that the application knows will not be accessed, and the hardware can understand how to filter across adjacent tiles, alleviating some of the pain experienced by developers who perform software tiling on their own.

But to provide a complete solution, something must be done to deal with the fact that, independent of whether tiling within a surface is supported, the maximum surface dimension is currently 16384 - nowhere near the 128K+ that applications already want. Just requiring the hardware to support larger texture sizes is one approach, however there are significant costs and/or tradeoffs to going this route. Direct3D 11's texture filter path and rendering path are already saturated in terms of precision in supporting 16K textures with the other requirements, such as supporting viewport extents falling off the surface during rendering, or supporting

texture wrapping off the surface edge during filtering. A possibility is to define a tradeoff such that as the texture size increases beyond 16K, functionality/precision is given up in some manner. Even with this concession however, additional hardware costs might be required in terms of addressing capability throughout the hardware system to go to larger texture sizes.

One issue that comes into play as textures get very large is that single precision floating point texture coordinates (and the associated interpolators to support rasterization) run out of precision to specify locations on the surface accurately. Jittery texture filtering would ensue. One expensive option would be to require double precision interpolator support, though that could be overkill given a reasonable alternative.

An alternate name for tiled resources is "sparse texture." "Sparse" conveys both the tiled nature of the resources as well as perhaps the primary reason for tiling them - that not all of them are expected to be mapped at once. In fact, an application could conceivably author a tiled resource in which no data is authored for all regions+mips of the resource, intentionally. So, the content itself could be sparse, and the mapping of the content in GPU memory at a given time would be a subset of that (even more sparse).

Another scenario that could be served by tiled resources is enabling multiple resources of different dimensions/formats to share the same memory. Sometimes applications have exclusive sets of resources that are known not to be used at the same time, or resources that are created only for very brief use and then destroyed, followed by creation of other resources. A form of generality that can fall out of "tiled resources" is that it is possible to allow the user to point multiple different resources at the same (overlapping) memory. In other words, the creation and destruction of "resources" (which define a dimension/format and so on) can be decoupled from the management of the memory underlying the resources from the application's point of view.

Related topics

[Tiled resources](#)

Creating tiled resources

2/22/2020 • 2 minutes to read • [Edit Online](#)

Tiled resources are created by specifying the [D3D11_RESOURCE_MISC_TILED](#) flag when you create a resource.

Restrictions on when you can use [D3D11_RESOURCE_MISC_TILED](#) to create a resource are described in [Tiled resource creation parameters](#).

A non-tiled resource's storage is allocated in the graphics system when the resource is created. For example, when you call [ID3D11Device::CreateTexture2D](#) to create an array of 2D textures, the graphics system allocates storage for those 2D textures. When a tiled resource is created, the graphics system doesn't allocate the storage for the resource contents. Instead, when an application creates a tiled resource, the graphics system makes an address space reservation for the tiled surface's area only, and then allows the mapping of the tiles to be controlled by the application. The "mapping" of a tile is simply the physical location in memory that a logical tile in a resource points to (or `NULL` for an unmapped tile). Don't confuse this concept with the notion of mapping a Direct3D resource for CPU access, which despite using the same name is completely independent. You will be able to define and change the mapping of each tile individually as needed, knowing that all tiles for a surface don't need to be mapped at a time, thereby making effective use of the amount of memory available.

This section provides more info about how to create tiled resources.

In this section

TOPIC	DESCRIPTION
Mappings are into a tile pool	When a resource is created with the D3D11_RESOURCE_MISC_TILED flag, the tiles that make up the resource come from pointing at locations in a tile pool. A tile pool is a pool of memory (backed by one or more allocations behind the scenes - unseen by the application).
Tiled resource creation parameters	There are some constraints on the type of Direct3D resources that you can create with the D3D11_RESOURCE_MISC_TILED flag. This section provides the valid parameters for creating tiled resources.
Address space available for tiled resources	This section specifies the virtual address space that is available for tiled resources.
Tile pool creation parameters	Use the parameters in this section to define tile pools via the ID3D11Device::CreateBuffer API.
Tiled resource cross process and device sharing	Tile pools can be shared with other processes just like traditional resources. Tiled resources that reference tile pools can't be shared across devices and processes. But separate processes can create their own tiled resources that map to tile pools that are shared between those tiled resources.
Operations available on tiled resources	This section lists operations that you can perform on tiled resources.

TOPIC	DESCRIPTION
Operations available on tile pools	This section lists operations that you can perform on tile pools.
How a tiled resource's area is tiled	When you create a tiled resource, the dimensions, format element size, and number of mipmaps and/or array slices (if applicable) determine the number of tiles that are required to back the entire surface area.

Related topics

[Tiled resources](#)

Mappings are into a tile pool

11/2/2020 • 4 minutes to read • [Edit Online](#)

When a resource is created with the [D3D11_RESOURCE_MISC_TILED](#) flag, the tiles that make up the resource come from pointing at locations in a tile pool. A tile pool is a pool of memory (backed by one or more allocations behind the scenes - unseen by the application). The operating system and display driver manage this pool of memory, and the memory footprint is easily understood by an application. Tiled resources map 64KB regions by pointing to locations in a tile pool. One fallout of this setup is it allows multiple resources to share and reuse the same tiles, and also for the same tiles to be reused at different locations within a resource if desired.

The cost for the flexibility of populating the tiles for a resource out of a tile pool is that the resource has to do the work of defining and maintaining the mapping of which tiles in the tile pool represent the tiles needed for the resource. Tile mappings can be changed. Also, not all tiles in a resource need to be mapped at a time; a resource can have **NULL** mappings. A **NULL** mapping defines a tile as not being available from the point of view of the resource accessing it.

Multiple tile pools can be created, and any number of tiled resources can map into any given tile pool at the same time. Tile pools can also be grown or shrunk. For more info, see [Tile pool resizing](#). One constraint that exists to simplify display driver and runtime implementation is that a given tiled resource can only have mappings into at most one tile pool at a time (as opposed to having simultaneous mapping to multiple tile pools).

The amount of storage that is associated with a tiled resource itself (that is, independent tile-pool memory) is roughly proportional to the number of tiles actually mapped to the pool at any given time. In hardware, this fact boils down to scaling the memory footprint for page table storage roughly with the amount of tiles that are mapped (for example, using a multilevel page table scheme as appropriate).

The tile pool can be thought of as an entirely software abstraction that enables Direct3D applications to effectively be able to program the page tables on the graphics processing unit (GPU) without having to know the low level implementation details (or deal with pointer addresses directly). Tile pools don't apply any additional levels of indirection in hardware. Optimizations of a single level page table using constructs like page directories are independent of the tile pool concept.

Let us explore what storage the page table itself could require in the worst case (though in practice implementations only require storage roughly proportional to what is mapped).

Suppose each page table entry is 64 bits.

For the worst-case page table size hit for a single surface, given the resource limits in Direct3D 11, suppose a tiled resource is created with a 128 bit-per-element format (for example, a RGBA float), so a 64KB tile contains only 4096 pixels. The maximum supported [Texture2DArray](#) size of 16384*16384*2048 (but with only a single mipmap) would require about 1GB of storage in the page table if fully populated (not including mipmaps) using 64 bit table entries. Adding mipmaps would grow the fully-mapped (worst case) page table storage by about a third, to about 1.3GB.

This case would give access to about 10.6 terabytes of addressable memory. There might be a limit on the amount of addressable memory however, which would reduce these amounts, perhaps to around the terabyte range.

Another case to consider is a single [Texture2D](#) tiled resource of 16384*16384 with a 32 bit-per-element format, including mipmaps. The space needed in a fully populated page table would be roughly 170KB with 64 bit table

entries.

Finally, consider an example using a BC format, say BC7 with 128 bits per tile of 4x4 pixels. That is one byte per pixel. A [Texture2DArray](#) of 16384*16384*2048 including mipmaps would require roughly 85MB to fully populate this memory in a page table. That is not bad considering this allows one tiled resource to span 550 gigapixels (512 GB of memory in this case).

In practice, nowhere near these full mappings would be defined given that the amount of physical memory available wouldn't allow anywhere near that much to be mapped and referenced at a time. But with a tile pool, applications could choose to reuse tiles (as a simple example, reusing a "black" colored tile for large black regions in an image) - effectively using the tile pool (that is, page table mappings) as a tool for memory compression.

The initial contents of the page table are **NULL** for all entries. Applications also can't pass initial data for the memory contents of the surface since it starts off with no memory backing.

In this section

TOPIC	DESCRIPTION
Tile pool creation	A tile pool is created via the ID3D11Device::CreateBuffer API by passing the D3D11_RESOURCE_MISC_TILE_POOL flag in the MiscFlags member of the D3D11_BUFFER_DESC structure that the <i>pDesc</i> parameter points to.
Tile pool resizing	Use the ID3D11DeviceContext2::ResizeTilePool API to grow a tile pool if the application needs more working set for the tiled resources mapping into it or to shrink if less space is needed.
Hazard tracking versus tile pool resources	For non-tiled resources, Direct3D can prevent certain hazard conditions during rendering, but because hazard tracking would be at a tile level for tiled resources, tracking hazard conditions during rendering of tiled resources might be too expensive.

Related topics

[Creating tiled resources](#)

Tile pool creation

2/22/2020 • 2 minutes to read • [Edit Online](#)

A tile pool is created via the [ID3D11Device::CreateBuffer](#) API by passing the [D3D11_RESOURCE_MISC_TILE_POOL](#) flag in the [MiscFlags](#) member of the [D3D11_BUFFER_DESC](#) structure that the *pDesc* parameter points to.

Applications can create one or more tile pools per Direct3D device. The total size of each tile pool is restricted to Direct3D 11's resource size limit, which is roughly 1/4 of graphics processing unit (GPU) RAM.

A tile pool is made of 64KB tiles, but the operating system (display driver) manages the entire pool as one or more allocations behind the scenes—the breakdown is not visible to applications. Tiled resources define content by pointing at tiles within a tile pool. Unmapping a tile from a tiled resource is done by pointing the tile to `NULL`. Such unmapped tiles have rules about the behavior of reads or writes. For info, see [Hazard tracking versus tile pool resources](#).

Related topics

[Mappings are into a tile pool](#)

Tile pool resizing

2/22/2020 • 2 minutes to read • [Edit Online](#)

Use the [ID3D11DeviceContext2::ResizeTilePool](#) API to grow a tile pool if the application needs more working set for the tiled resources mapping into it or to shrink if less space is needed. Another option for applications is to allocate additional tile pools for new tiled resources. But if any single tiled resource needs more space than initially available in its tile pool, growing the tile pool is a good option. A tiled resource can't have mappings into multiple tile pools at the same time.

When a tile pool is grown, additional tiles are added to the end via one or more new allocations by the display driver. This breakdown into allocations isn't visible to the application. Existing memory in the tile pool is left untouched, and existing tiled resource mappings into that memory remain intact.

When a tile pool is shrunk, tiles are removed from the end. Tiles are removed even below the initial allocation size, down to 0, which means new mappings can't be made past the new size. But, existing mappings past the end of the new size remain intact and useable. The display driver will keep the memory around as long as mappings to any part of the allocations that the driver uses for the tile pool memory remains. If after shrinking some memory has been kept alive because tile mappings are pointing to it and then the tile pool is regrown again (by any amount), the existing memory is reused first before any additional allocations occur to service the size of the grow operation.

To be able to save memory, an application has to not only shrink a tile pool but also remove/remap existing mappings past the end of the new smaller tile pool size.

The act of shrinking (and removing mappings) doesn't necessarily produce immediate memory savings. Freeing of memory depends on how granular the display driver's underlying allocations for the tile pool are. When shrinking happens to be enough to make a display driver allocation unused, the display driver can free it. If a tile pool was grown, shrinking to previous sizes (and removing/remapping tile mappings correspondingly) is most likely to yield memory savings, though not guaranteed in the case that the sizes don't exactly align with the underlying allocation sizes chosen by the display driver.

Related topics

[Mappings are into a tile pool](#)

Hazard tracking versus tile pool resources

2/22/2020 • 2 minutes to read • [Edit Online](#)

For non-tiled resources, Direct3D can prevent certain hazard conditions during rendering, but because hazard tracking would be at a tile level for tiled resources, tracking hazard conditions during rendering of tiled resources might be too expensive.

For example, for non-tiled resources, the runtime doesn't allow any given SubResource to be bound as an input (such as a ShaderResourceView) and as an output (such as a RenderTargetView) at the same time. If such a case is encountered, the runtime unbinds the input. This tracking overhead in the runtime is cheap and is done at the SubResource level. One of the benefits of this tracking overhead is to minimize the chances of applications accidentally depending on hardware shader execution order. Hardware shader execution order can vary if not on a given graphics processing unit (GPU), then certainly across different GPUs.

Tracking how resources are bound might be too expensive for tiled resources because tracking is at a tile level. New issues arise such as possibly validating away attempts to render to an RenderTargetView with one tile mapped to multiple areas in the surface simultaneously. If it turns out this per-tile hazard tracking is too expensive for the runtime, ideally this would at least be an option in the debug layer.

An application must inform the display driver when it has issued a write or read operation to a tiled resource that references tile pool memory that will also be referenced by separate tiled resources in upcoming read or write operations that it is expecting the first operation to complete before the following operations can begin. For more info about this condition, see [ID3D11DeviceContext2::TiledResourceBarrier](#) remarks.

Related topics

[Mappings are into a tile pool](#)

Tiled resource creation parameters

2/22/2020 • 2 minutes to read • [Edit Online](#)

There are some constraints on the type of Direct3D resources that you can create with the [D3D11_RESOURCE_MISC_TILED](#) flag. This section provides the valid parameters for creating tiled resources.

Supported Resource Type

Texture2D[Array] (including TextureCube[Array], which is a variant of Texture2D[Array]) or Buffer.

NOT supported: Texture1D[Array] or Texture3D, but Texture3D might be supported in the future.

Supported Resource Usage

D3D11_USAGE_DEFAULT.

NOT supported: D3D11_USAGE_DYNAMIC, D3D11_USAGE_STAGING, or D3D11_USAGE_IMMUTABLE.

Supported Resource Misc Flags

D3D11_RESOURCE_MISC_TILED (by definition), _MISC_TEXTURECUBE, _DRAWINDIRECT_ARGS, _BUFFER_ALLOW_RAW_VIEWS, _BUFFER_STRUCTURED, _RESOURCE_CLAMP, or _GENERATE_MIPS.

NOT supported: _SHARED, _SHARED_KEYEDMUTEX, _GDI_COMPATIBLE, _SHARED_NTHANDLE, _RESTRICTED_CONTENT, _RESTRICT_SHARED_RESOURCE, _RESTRICT_SHARED_RESOURCE_DRIVER, _GUARDED, or _TILE_POOL.

Supported Bind Flags

D3D11_BIND_SHADER_RESOURCE, _RENDER_TARGET, _DEPTH_STENCIL, or _UNORDERED_ACCESS.

NOT supported: _CONSTANT_BUFFER, _VERTEX_BUFFER [note that binding a tiled Buffer as an SRV/UAV/RTV is still ok], _INDEX_BUFFER, _STREAM_OUTPUT, _BIND_DECODER, or _BIND_VIDEO_ENCODER.

Supported Formats

All formats that would be available for the given configuration regardless of it being tiled, with some exceptions.

Supported SampleDesc (Multisample count, quality)

Whatever would be supported for the given configuration regardless of it being tiled, with some exceptions.

Supported Width/Height/MipLevels/ArraySize

Full extents supported by Direct3D 11. Tiled resources don't have the restriction on total memory size imposed on non-tiled resources. Tiled resources are only constrained by overall virtual address space limits. For info, see [Address space available for tiled resources](#).

The initial contents of tile pool memory are undefined.

Related topics

[Creating tiled resources](#)

Address space available for tiled resources

2/22/2020 • 2 minutes to read • [Edit Online](#)

This section specifies the virtual address space that is available for tiled resources.

On 64-bit operating systems, at least 40 bits of virtual address space (1 Terabyte) is available.

For 32-bit operating systems, the address space is 32 bit (4 GB). For 32-bit ARM systems, individual tiled resource creation can fail if the allocation would use more than 27 bits of address space (128 MB). This includes any hidden padding in the address space the hardware may use for mipmaps, packed tile padding, and possibly padding surface dimensions to powers of 2.

On graphics systems with a separate page table for the graphics processing unit (GPU), most of this address space will be available to GPU resources made by the application, though GPU allocations made by the display driver fit in the same space.

On future systems with a page table shared between the CPU and GPU, the available address space is shared between all CPU and GPU allocations in a process.

Related topics

[Tiled resource creation parameters](#)

Tile pool creation parameters

2/22/2020 • 2 minutes to read • [Edit Online](#)

Use the parameters in this section to define tile pools via the [ID3D11Device::CreateBuffer](#) API.

Size

Allocation size, as a multiple of 64KB. 0 is valid because you can later use a [ID3D11DeviceContext2::ResizeTilePool](#) operation.

Supported Resource Misc Flags

D3D11_RESOURCE_MISC_TILE_POOL (identifies the resource as a tile pool), D3D11_RESOURCE_MISC_SHARED,_SHARED_KEYEDMUTEX, or _SHARED_NTHANDLE.

Supported Resource Usage

D3D11_USAGE_DEFAULT only.

Related topics

[Creating tiled resources](#)

Tiled resource cross process and device sharing

2/22/2020 • 2 minutes to read • [Edit Online](#)

Tile pools can be shared with other processes just like traditional resources. Tiled resources that reference tile pools can't be shared across devices and processes. But separate processes can create their own tiled resources that map to tile pools that are shared between those tiled resources.

Shared tile pools can't be resized.

In this section

TOPIC	DESCRIPTION
Stencil formats not supported with tiled resources	Formats that contain stencil aren't supported with tiled resources.

Related topics

[Creating tiled resources](#)

Stencil formats not supported with tiled resources

2/22/2020 • 2 minutes to read • [Edit Online](#)

Formats that contain stencil aren't supported with tiled resources.

Formats that contain stencil include DXGI_FORMAT_D24_UNORM_S8_UINT (and related formats in the R24G8 family) and DXGI_FORMAT_D32_FLOAT_S8X24_UINT (and related formats in the R32G8X24 family).

Some implementations store depth and stencil in separate allocations while others store them together. Tile management for the two schemes would have to be different, and no single API can abstract or rationalize the differences. We recommend for future hardware to support independent depth and stencil surfaces, each independently tiled. 32 bit depth would have 128x128 tiles, and 8 bit stencil would have 256x256 tiles.

Therefore, applications would have to live with tile shape misalignment between depth and stencil. But the same problem exists with different render target surface formats already.

Related topics

[Tiled resource cross process and device sharing](#)

Operations available on tiled resources

2/22/2020 • 2 minutes to read • [Edit Online](#)

This section lists operations that you can perform on tiled resources.

- void [ID3D11DeviceContext2::UpdateTileMappings](#) and [ID3D11DeviceContext2::CopyTileMappings](#) operations - These operations point tile locations in a tiled resource to locations in tile pools, or to NULL, or to both. These operations can update a disjoint subset of the tile pointers.
- Copy*() and Update*() operations - All the APIs that can copy data to and from a default pool surface (for example, [ID3D11DeviceContext1::CopySubresourceRegion1](#) and [ID3D11DeviceContext1::UpdateSubresource1](#)) work for tiled resources. Reading from unmapped tiles produces 0 and writes to unmapped tiles are dropped.
- [ID3D11DeviceContext2::CopyTiles](#) and [ID3D11DeviceContext2::UpdateTiles](#) operations - These operations exist for copying tiles at 64KB granularity to and from any tiled resource and a buffer resource in a canonical memory layout. The display driver and hardware perform any memory "swizzling" necessary for the tiled resource.
- Direct3D pipeline bindings and view creations / bindings that would work on non-tiled resources work on tiled resources as well.

Tile controls are available on immediate or deferred contexts (just like updates to typical resources) and upon execution impact subsequent accesses to the tiles (not previously submitted operations).

Related topics

[Creating tiled resources](#)

Operations available on tile pools

11/2/2020 • 2 minutes to read • [Edit Online](#)

This section lists operations that you can perform on tile pools.

- The lifetime of tile pools works like any other Direct3D Resource, backed by reference counting, including in this case tracking of mappings from tiled resources. When the application no longer references a tile pool and any tile mappings to the memory are gone and graphics processing unit (GPU) accesses completed, the operating system will deallocate the tile pool.
- APIs related to surface sharing and synchronization work for tile pools (but not directly on tiled resources). Similar to the behavior for offered tile pools, Direct3D commands that access tiled resources that point to a tile pool are dropped if the tile pool has been shared and is currently acquired by another device and process.
- [ID3D11DeviceContext2::ResizeTilePool](#) operation
- [IDXGIDevice2::OfferResources](#) and [ReclaimResources](#) operations - These APIs for yielding memory temporarily to the system operate on the entire tile pool (and aren't available for individual tiled resources). If a tiled resource points to a tile in an offered tile pool, the tiled resource behaves as if it is offered (for example, the runtime drops commands that reference it).

Data can't be copied to and from tile pool memory directly. Accesses to the memory are always done through tiled resources.

Related topics

[Creating tiled resources](#)

How a tiled resource's area is tiled

11/2/2020 • 2 minutes to read • [Edit Online](#)

When you create a tiled resource, the dimensions, format element size, and number of mipmaps and/or array slices (if applicable) determine the number of tiles that are required to back the entire surface area. The pixel/byte layout within tiles is determined by the implementation. The number of pixels that fit in a tile, depending on the format element size, is fixed and identical whether you use a standard swizzle or not.

The number of tiles that will be used by a given surface size and format element width is well defined and predictable based on the tables in the following sections. For resources that contain mipmaps or cases where surface dimensions don't fill a tile, some constraints exist and are discussed in [Mipmap packing](#).

Different tiled resources can point to identical memory with different formats as long as applications don't rely on the results of writing to the memory with one format and reading with another. But, in constrained circumstances, applications can rely on the results of writing to the memory with one format and reading with another if the formats are in the same format family (that is, they have the same typeless parent format). For example, DXGI_FORMAT_R8G8B8A8_UNORM and DXGI_FORMAT_R8G8B8A8_UINT are compatible with each other but not with DXGI_FORMAT_R16G16_UNORM. An application must conservatively match all resource properties in order to reinterpret data between two textures since tile patterns are very conservatively undefined. Obviously, though, the format is more lax. The formats need only be compatible as illustrated above. An exception is where bleeding data from one format aliasing to another is well defined: if a tile completely contains 0 for all its bits, that tile can be used with any format that interprets those memory contents as 0 (regardless of memory layout). So, a tile could be cleared to 0x00 with the format DXGI_FORMAT_R8_UNORM and then used with a format like DXGI_FORMAT_R32G32_FLOAT and it would appear the contents are still (0.0f,0.0f).

The layout of data within a tile may be depend on resource properties, the subresource in which it resides, and the tile location within the subresource. The major exceptions are illustrated above: compatible formats with other resource properties being equal and when all texels are the same pattern.

In this section

TOPIC	DESCRIPTION
Texture2D and Texture2DArray subresource tiling	These tables show how Texture2D and Texture2DArray subresources are tiled.
Texture3D subresource tiling	This table shows how Texture3D subresources are tiled.
Buffer tiling	A Buffer resource is divided into 64KB tiles, with some empty space in the last tile if the size is not a multiple of 64KB.
Mipmap packing	Depending on the tier of tiled resources support, mipmaps with certain dimensions don't follow the standard tile shapes and are considered to all be packed together with one another in a manner that is opaque to the application.

Related topics

[Creating tiled resources](#)

Texture2D and Texture2DArray subresource tiling

11/2/2020 • 2 minutes to read • [Edit Online](#)

These tables show how **Texture2D** and **Texture2DArray** subresources are tiled. The values in these tables don't count tail mip packing.

This table shows how **Texture2D** and **Texture2DArray** subresources with multisample counts of 1 are tiled.

BITS/PIXEL (1 SAMPLE/PIXEL)	TILE DIMENSIONS (PIXELS, WXH)
8	256x256
16	256x128
32	128x128
64	128x64
128	64x64
BC1,4	512x256
BC2,3,5,6,7	256x256

Format bit counts not supported with tiled resources are 96 bpp formats, video formats, DXGI_FORMAT_R1_UNORM, DXGI_FORMAT_R8G8_B8G8_UNORM, and DXGI_FORMAT_R8R8_G8B8_UNORM.

This table shows how **Texture2D** and **Texture2DArray** subresources with various multisample counts are tiled.

MULTISAMPLE COUNT	DIVIDE TILE DIMENSIONS ABOVE BY (WXH)
1	1x1
2	2x1
4	2x2
8	4x2
16	4x4

Only sample counts 1 and 4 are required (and allowed) to be supported with tiled resources. Tiled resources don't currently support 2, 8, and 16 even though they are shown.

Implementations can choose to support 2, 8, and/or 16 sample multisample antialiasing (MSAA) mode for non-tiled resources even though tiled resource don't support them.

Tiled resources with sample counts larger than 1 can't use 128 bpp formats.

Related topics

[How a tiled resource's area is tiled](#)

Texture3D subresource tiling

11/2/2020 • 2 minutes to read • [Edit Online](#)

This table shows how **Texture3D** subresources are tiled. The values in this table don't count tail mip packing.

This table takes the **Texture2D** tiling and divides the x/y dimensions by 4 each and adds 16 layers of depth. All the tiles for the first plane (2D plane of tiles defining the first 16 layers of depth) appear before the subsequent planes.

NOTE

Texture3D support in tiled resources isn't exposed in the initial implementation of tiled resources, but the desired tile shapes are listed here because support might be consideration in a future release.

BITS/PIXEL (1 SAMPLE/PIXEL)	TILE DIMENSIONS (PIXELS, WXHxD)
8	64x32x32
16	32x32x32
32	32x32x16
64	32x16x16
128	16x16x16
BC1,4	128x64x16
BC2,3,5,6,7	64x64x16

Format bit counts not supported with tiled resources are 96 bpp formats, video formats, DXGI_FORMAT_R1_UNORM, DXGI_FORMAT_R8G8_B8G8_UNORM, and DXGI_FORMAT_R8R8_G8B8_UNORM.

Related topics

[How a tiled resource's area is tiled](#)

Buffer tiling

11/2/2020 • 2 minutes to read • [Edit Online](#)

A [Buffer](#) resource is divided into 64KB tiles, with some empty space in the last tile if the size is not a multiple of 64KB.

Structured buffers must have no constraint on the stride to be tiled. But possible performance optimizations in hardware for using [StructuredBuffers](#) can be sacrificed by making them tiled in the first place.

Related topics

[How a tiled resource's area is tiled](#)

Mipmap packing

2/22/2020 • 2 minutes to read • [Edit Online](#)

Depending on the [tier](#) of tiled resources support, mipmaps with certain dimensions don't follow the standard tile shapes and are considered to all be packed together with one another in a manner that is opaque to the application. Higher tiers of support have broader guarantees about what types of surface dimensions fit in the standard tile shapes (and can therefore be individually mapped by applications).

What can vary between implementations is that—given a tiled resource's dimensions, format, number of mipmaps, and array slices—some number M of mips (per array slice) can be packed into some number N tiles. The [ID3D11Device2::GetResourceTiling](#) API exists to allow the driver to report to the application what M and N are (among other details about the surface that this API reports that are standard and don't vary by hardware vendor). The set of tiles for the packed mips are still 64KB and can be individually mapped into disparate locations in a tile pool. But the pixel shape of the tiles and how the mipmaps fit across the set of tiles is specific to a hardware vendor and too complex to expose. So, applications are required to either map all of the tiles that are designated as packed, or none of them, at a time. Otherwise, the behavior for accessing the tiled resource is undefined.

For arrayed surfaces, the set of packed mips and the number of packed tiles storing those mips (M and N described preceding) applies individually for each array slice.

Dedicated APIs for copying tiles ([ID3D11DeviceContext2::CopyTiles](#) and [ID3D11DeviceContext2::UpdateTiles](#)) can't access packed mips. Applications that want to copy data to and from packed mips can do so using all the non-tiled resource specific APIs for copying and rendering to surfaces.

Related topics

[How a tiled resource's area is tiled](#)

Tiled Resource APIs

2/22/2020 • 4 minutes to read • [Edit Online](#)

The APIs described in this section work with tiled resources and tile pool.

- [Assigning tiles from a tile pool to a resource](#)
- [Querying resource tiling and support](#)
- [Copying tiled data](#)
- [Resizing tile pool](#)
- [Tiled resource barrier](#)
- [Related topics](#)

Assigning tiles from a tile pool to a resource

The [ID3D11DeviceContext2::UpdateTileMappings](#) and [ID3D11DeviceContext2::CopyTileMappings](#) APIs manipulate and query tile mappings. Update calls only affect the tiles identified in the call, and others are left as defined previously.

Any given tile from a tile pool can be mapped to multiple locations in a resource and even multiple resources. This mapping includes tiles in a resource that have an implementation-chosen layout ([Mipmap packing](#)) where multiple mipmaps are packed together into a single tile. The catch is that if data is written to the tile via one mapping, but read via a differently configured mapping, the results are undefined. Careful use of this flexibility can still be useful for an application though, like sharing a tile between resources that will not be used simultaneously, where the contents of the tile are always initialized through the same resource mapping as they will be subsequently read from. Similarly, a tile mapped to hold the packed mipmaps of multiple different resources with the same surface dimensions will work fine - the data will appear the same in both mappings.

Changes to tile assignments for a resource can be made at any time in an immediate or deferred context.

Querying resource tiling and support

To query resource tiling, use [ID3D11Device2::GetResourceTiling](#).

For other resource tiling support, use [ID3D11Device2::CheckMultisampleQualityLevels1](#).

Copying tiled data

Any methods in Direct3D for moving data around work with tiled resources just as if they are not tiled, except that writes to unmapped areas are dropped and reads from unmapped areas produce 0. If a copy operation involves writing to the same memory location multiple times because multiple locations in the destination resource are mapped to the same tile memory, the resulting writes to multi-mapped tiles are non-deterministic and non-repeatable. That is, accesses happen in whatever order the hardware happens to execute the copy.

Direct3D 11.2 introduces methods for these additional ways to copy:

- Copy between tiles in a tiled resource (at 64KB tile granularity) and (to/from) a buffer in graphics processing unit (GPU) memory (or staging resource) - [ID3D11DeviceContext2::CopyTiles](#)
- Copy from application-provided memory to tiles in a tiled resource - [ID3D11DeviceContext2::UpdateTiles](#)

These methods swizzle/deswizzle as needed, and allow a D3D11_TILE_COPY_NO_OVERWRITE flag when the

caller promises the destination memory is not referenced by GPU work that is in flight.

The tiles involved in the copy can't include tiles that contain packed mipmaps or that have results that are undefined. To transfer data to/from mipmaps that the hardware packs into one tile, you must use the standard (non-tile specific) Copy/Update APIs or [ID3D11DeviceContext::GenerateMips](#) for the whole mip chain.

Note on GenerateMips: Using [ID3D11DeviceContext::GenerateMips](#) on a resource with partially mapped tiles will produce results that simply follow the rules for reading and writing **NULL** applied to whatever algorithm the hardware and display driver happen to use to **GenerateMips**. So, it is not particularly useful for an application to bother doing this unless somehow the areas with **NULL** mappings (and their effect on other mips during the generation phase) will have no consequence on the parts of the surface the application does care about.

Copying tile data from a staging surface or from application memory would be the way to upload tiles that may have been streamed off disk, for example. A variation when streaming off disk is uploading some sort of compressed data to GPU memory and then decoding on the GPU. The decode target could be a buffer resource in GPU memory, from which [CopyTiles](#) then copies to the actual tiled resource. This copy step allows the GPU to swizzle when the swizzle pattern is not known. Swizzling is not needed if the tiled resource itself is a buffer resource (for example, as opposed to a Texture).

The memory layout of the tiles in the non-tiled buffer resource side of the copy is simply linear in memory within 64KB tiles, which the hardware and display driver would swizzle/deswizzle per tile as appropriate when transferring to/from a tiled resource. For multisample antialiasing (MSAA) surfaces, each pixel's samples are traversed in sample-index order before moving to the next pixel. For tiles that are partially filled on the right side (for a surface that has a width not a multiple of tile width in pixels), the pitch/stride to move down a row is the full size in bytes of the number pixels that would fit across the tile if the tile was full. So, there can be a gap between each row of pixels in memory. For specification simplicity, mipmaps smaller than a tile are not packed together in the linear layout. This seems to be a waste of memory space, but as mentioned copying to mips that the hardware packs together is not allowed via [CopyTiles](#) or [UpdateTiles](#). The application can just use generic [UpdateSubresource*](#)() or [CopySubresource*](#)() APIs to copy small mips individually, though in the case of [CopySubresource*](#)() that means the linear memory has to be the same dimension as the tiled resource - [CopySubresource*](#)() can't copy from a buffer resource to a Texture2D for instance.

If a hardware standard swizzle is defined, flags could be added to indicate that the data in the buffer is to be interpreted in that format (no swizzle necessary on transfer), though alternative approaches to uploading data may also make sense in that case such as allowing applications direct access to tile pool memory.

Copying operations can be done on an immediate or deferred context.

Resizing tile pool

To resize a tile pool, use [ID3D11DeviceContext2::ResizeTilePool](#).

Tiled resource barrier

To specify a data access ordering constraint between multiple tiled resources, use [ID3D11DeviceContext2::TiledResourceBarrier](#).

Related topics

[Tiled resources](#)

Pipeline access to tiled resources

11/2/2020 • 2 minutes to read • [Edit Online](#)

Tiled resources can be used in shader resource views (SRV), render target views (RTV), depth stencil views (DSV) and unordered access views (UAV), as well as some bind points where views aren't used, such as vertex buffer bindings. For the list of supported bindings, see [Tiled resource creation parameters](#). Copy* operations also work on tiled resources.

If multiple tile coordinates in one or more views is bound to the same memory location, reads and writes from different paths to the same memory will occur in a non-deterministic and non-repeatable order of memory accesses.

If all tiles behind a memory access footprint from a shader are mapped to unique tiles, behavior is identical on all implementations to the surface having the same memory contents in a non-tiled fashion.

This section provides more info about pipeline access to tiled resources.

In this section

TOPIC	DESCRIPTION
SRV behavior with non-mapped tiles	Behavior of shader resource view (SRV) reads that involve non-mapped tiles depends on the level of hardware support.
UAV behavior with non-mapped tiles	Behavior of unordered access view (UAV) reads and writes depends on the level of hardware support.
Rasterizer behavior with non-mapped tiles	This section describes rasterizer behavior with non-mapped tiles.
Tile access limitations with duplicate mappings	This section describes tile access limitations with duplicate mappings.
Tiled resources texture sampling features	This section describes tiled resources texture sampling features.
HLSL tiled resources exposure	New Microsoft High Level Shader Language (HLSL) syntax is required to support tiled resources in Shader Model 5 .

Related topics

[Tiled resources](#)

SRV behavior with non-mapped tiles

2/22/2020 • 2 minutes to read • [Edit Online](#)

Behavior of shader resource view (SRV) reads that involve non-mapped tiles depends on the level of hardware support. For a breakdown of requirements, see read behavior for [Tiled resources features tiers](#). This section summarizes the ideal behavior, which [Tier 2](#) requires.

Consider a texture filter operation that reads from a set of texels in an SRV. Texels that fall on non-mapped tiles contribute 0 in all non-missing components of the format (and the default for missing components) into the overall filter operation alongside contributions from mapped texels. The texels are all weighted and combined together independent of whether the data came from mapped or non-mapped tiles.

Some first generation [Tier 2](#) level hardware does not meet this spec requirement and returns the 0 with defaults described preceding as the overall filter result if any texels (with nonzero weight) fall on non-mapped tiles. No other hardware will be allowed to miss the requirement to include all (nonzero weight) texels in the filter.

Related topics

[Pipeline access to tiled resources](#)

UAV behavior with non-mapped tiles

2/22/2020 • 2 minutes to read • [Edit Online](#)

Behavior of unordered access view (UAV) reads and writes depends on the level of hardware support. For a breakdown of requirements, see overall read and write behavior for [Tiled resources features tiers](#). This section summarizes the ideal behavior.

Shader operations that read from a non-mapped tile in a UAV return 0 in all non-missing components of the format, and the default for missing components.

Shader operations that attempt to write to a non-mapped tile cause nothing to be written to the non-mapped area (while writes to a mapped area proceed). This ideal definition for write handling is not required by [Tier 2](#); writes to non-mapped tiles can end up in a cache that subsequent reads could pick up.

Related topics

[Pipeline access to tiled resources](#)

Rasterizer behavior with non-mapped tiles

2/22/2020 • 2 minutes to read • [Edit Online](#)

This section describes rasterizer behavior with non-mapped tiles.

DepthStencilView

Behavior of depth stencil view (DSV) reads and writes depends on the level of hardware support. For a breakdown of requirements, see overall read and write behavior for [Tiled resources features tiers](#).

Here is the ideal behavior:

If a tile isn't mapped in the DepthStencilView, the return value from reading depth is 0, which is then fed into whatever operations are configured for the depth read value. Writes to the missing depth tile are dropped. This ideal definition for write handling is not required by [Tier 2](#); writes to non-mapped tiles can end up in a cache that subsequent reads could pick up.

RenderTargetView

Behavior of render target view (RTV) reads and writes depends on the level of hardware support. For a breakdown of requirements, see overall read and write behavior for [Tiled resources features tiers](#).

On all implementations, different RTVs (and DSV) bound simultaneously can have different areas mapped versus non-mapped and can have different sized surface formats (which means different tile shapes).

Here is the ideal behavior:

Reads from RTVs return 0 in missing tiles and writes are dropped. This ideal definition for write handling is not required by [Tier 2](#); writes to non-mapped tiles can end up in a cache that subsequent reads could pick up.

Related topics

[Pipeline access to tiled resources](#)

Tile access limitations with duplicate mappings

2/22/2020 • 5 minutes to read • [Edit Online](#)

This section describes tile access limitations with duplicate mappings.

Copying tiled resources with overlapping source and destination

If tiles in the source and destination area of a Copy* operation have duplicated mappings in the copy area that would have overlapped even if both resources were not tiled resources and the Copy* operation supports overlapping copies, the Copy* operation will behave fine (as if the source is copied to a temporary location before going to the destination). But if the overlap is not obvious (like the source and destination resources are different but share mappings or mappings are duplicated over a given surface), results of the copy operation on the tiles that are shared are undefined.

Copying to tiled resource with duplicated tiles in destination area

Copying to a tiled resource with duplicated tiles in the destination area produces undefined results in these tiles unless the data itself is identical; different tiles might write the tiles in different orders.

UAV accesses to duplicate tiles mappings

Suppose an unordered access view (UAV) on a tiled resource has duplicate tile mappings in its area or with other resources bound to the pipeline. Ordering of accesses to these duplicated tiles is undefined if performed by different threads, just as any ordering of memory access to UAVs in general is unordered.

Rendering after tile mapping changes or content updates from outside mappings

If a tiled resource's tile mappings have changed or content in mapped tiled pool tiles have changed via another tiled resource's mappings, and the tiled resource is going to be rendered via render target view or depth stencil view, the application must Clear (using the fixed function Clear APIs) or fully copy over using Copy*/Update* APIs the tiles that have changed within the area being rendered (mapped or not). Failure of an application to clear or copy in these cases results in hardware optimization structures for the given render target view or depth stencil view being stale and will result in garbage rendering results on some hardware and inconsistency across different hardware. These hidden optimization data structures used by hardware might be local to individual mappings and not visible to other mappings to the same memory.

The [ID3D11DeviceContext1::ClearView](#) operation supports clearing render target views with rectangles. For hardware that supports tiled resources, [ClearView](#) must also support clearing of depth stencil views with rectangles, for depth only surfaces (without stencil). This operation allows applications to clear only the necessary area of a surface.

If an application needs to preserve existing memory contents of areas in a tiled resource where mappings have changed, that application must work around the clear requirement. The application can accomplish this work-around by first saving the contents where tile mappings have changed (by copying them to a temporary surface, for example, by using [ID3D11DeviceContext2::CopyTiles](#)), issuing the required clear command and then copying the contents back. While this would accomplish the task of preserving surface contents for incremental rendering, the downside is that subsequent rendering performance on the surface might suffer because rendering optimizations might be lost.

If a tile is mapped into multiple tiled resources at the same time and tile contents are manipulated by any means (render, copy, and so on) via one of the tiled resources, if the same tile is to be rendered via any other tiled resource, the tile must be cleared first as previously described.

Rendering to tiles shared outside render area

Suppose an area in a tiled resource is being rendered to and the tile pool tiles referenced by the render area are also mapped to from outside the render area (including via other tiled resources, at the same time or not). Data rendered to these tiles isn't guaranteed to appear correctly when viewed through the other mappings, even though the underlying memory layout is compatible. This fact is due to optimization data structures some hardware use that can be local to individual mappings for renderable surfaces and not visible to other mappings to the same memory location. You can work around this restriction by copying from the rendered mapping to all the other mappings to the same memory that might be accessed (or clearing that memory or copying other data to it if the old contents are no longer needed). While this work-around seems redundant, it makes all other mappings to the same memory correctly understand how to access its contents, and at least the memory savings of having only a single physical memory backing remains intact. Also, when you switch between using different tiled resources that share mappings (unless only reading), you must call the [ID3D11DeviceContext2::TiledResourceBarrier](#) API in between the switches.

Rendering to tiles shared within render area

If an area in a tiled resource is being rendered to and within the render area multiple tiles are mapped to the same tile pool location, rendering results are undefined on those tiles.

Data compatibility across tiled resources sharing tiles

Suppose multiple tiled resources have mappings to the same tile pool locations and each resource is used to access the same data. This scenario is only valid if the other rules about avoiding problems with hardware optimization structures are avoided, appropriate calls to [ID3D11DeviceContext2::TiledResourceBarrier](#) are made, and the tiled resources are compatible with each other. The latter is described here in terms of what it means for tiled resources sharing tiles to be incompatible. The incompatibility conditions of accessing the same data across duplicate tile mappings are the use of different surface dimensions or format, or differences in the presence of render target or depth stencil bind flags on the resources. Writing to the memory with one type of mapping produces undefined results if you subsequently read or render via a mapping from an incompatible resource. If the other resource sharing mappings are first initialized with new data (recycling the memory for a different purpose), the subsequent read or render operation is fine since data isn't bleeding across incompatible interpretations. But, you must call the [TiledResourceBarrier](#) API when you switch between accessing incompatible mappings like this.

If the render target or depth stencil bind flag isn't set on any of the resources sharing mappings with each other, there are far fewer restrictions. As long as the format and surface types (for example, Texture2D) are the same, tiles can be shared. Different formats being compatible are cases such as BC* surfaces and the equivalent sized uncompressed 32 bit or 16 bit per component format, like BC6H and R32G32B32A32. Many 32 bit per element formats can be aliased with R32_* as well (R10G10B10A2_*, R8G8B8A8_*, B8G8R8A8_*, B8G8R8X8_*, R16G16_*); this operation has always been allowed for non-tiled resources.

Sharing between packed and non-packed tiles is fine if the formats are compatible and the tiles are filled with solid color.

Finally, if nothing is common about the resources sharing tile mappings except that none have render target or depth stencil bind flags, only memory filled with 0 can be shared safely; the mapping will appear as whatever 0 decodes to for the definition of the given resource format (typically just 0).

Related topics

[Pipeline access to tiled resources](#)

Tiled resources texture sampling features

11/2/2020 • 5 minutes to read • [Edit Online](#)

This section describes tiled resources texture sampling features.

Requirements of tiled resources texture sampling features

The texture sampling features described here require [Tier 2](#) level of tiled resources support.

Shader feedback about mapped areas

Any shader instruction that reads and/or writes to a tiled resource causes status information to be recorded. This status is exposed as an optional extra return value on every resource access instruction that goes into a 32-bit temp register. The contents of the return value are opaque. That is, direct reading by the shader program is disallowed. But, you can use the [CheckAccessFullyMapped](#) function to extract the status info.

Fully mapped check

The [CheckAccessFullyMapped](#) function interprets the status returned from a memory access and indicates whether all data being accessed was mapped in the resource. [CheckAccessFullyMapped](#) returns true (0xFFFFFFFF) if data was mapped or false (0x00000000) if data was unmapped.

During filter operations, sometimes the weight of a given texel ends up being 0.0. An example is a linear sample with texture coordinates that fall directly on a texel center: 3 other texels (which ones they are can vary by hardware) contribute to the filter but with 0 weight. These 0 weight texels don't contribute to the filter result at all, so if they happen to fall on **NULL** tiles, they don't count as an unmapped access. Note the same guarantee applies for texture filters that include multiple mip levels; if the texels on one of the mipmaps isn't mapped but the weight on those texels is 0, those texels don't count as an unmapped access.

When sampling from a format that has fewer than 4 components (such as `DXGI_FORMAT_R8_UNORM`), any texels that fall on **NULL** tiles result in the a **NULL** mapped access being reported regardless of which components the shader actually looks at in the result. For example, reading from `R8_UNORM` and masking the read result in the shader with `.gba/yzw` wouldn't appear to need to read the texture at all. But if the texel address is a **NULL** mapped tile, the operation still counts as a **NULL** map access.

The shader can check the status and pursue any desired course of action on failure. For example, a course of action can be logging 'misses' (say via UAV write) and/or issuing another read clamped to a coarser LOD known to be mapped. An application might want to track successful accesses as well in order to get a sense of what portion of the mapped set of tiles got accessed.

One complication for logging is no mechanism exists for reporting the exact set of tiles that would have been accessed. The application can make conservative guesses based on knowing the coordinates it used for access, as well as using the LOD instruction (for example, [tex2Dlod](#)) which returns what the hardware LOD calculation is.

Another complication is that lots of accesses will be to the same tiles, so a lot of redundant logging will occur and possibly contention on memory. It could be convenient if the hardware could be given the option to not bother to report tile accesses if they were reported elsewhere before. Perhaps the state of such tracking could be reset from the API (likely at frame boundaries).

Per-sample MinLOD clamp

To help shaders avoid areas in mipmapped tiled resources that are known to be non-mapped, most shader instructions that involve using a sampler (filtering) have a new mode that allows the shader to pass an additional float32 MinLOD clamp parameter to the texture sample. This value is in the view's mipmap number space, as opposed to the underlying resource.

The hardware performs `max(fShaderMinLODClamp,fComputedLOD)` in the same place in the LOD calculation where the per-resource MinLOD clamp occurs, which is also a `max()`.

If the result of applying the per-sample LOD clamp and any other LOD clamps defined in the sampler is an empty set, the result is the same out of bounds access result as the per-resource minLOD clamp: 0 for components in the surface format and defaults for missing components.

The LOD instruction (for example, `tex2Dlod`), which predates the per-sample minLOD clamp described here, returns both a clamped and unclamped LOD. The clamped LOD returned from this LOD instruction reflects all clamping including the per-resource clamp, but not a per-sample clamp. Per-sample clamp is controlled and known by the shader anyway, so the shader author can manually apply that clamp to the LOD instruction's return value if desired.

Min/Max reduction filtering

Applications can choose to manage their own data structures that inform them of what the mappings looks like for a tiled resource. An example would be a surface that contains a texel to hold information for every tile in a tiled resource. One might store the first LOD that is mapped at a given tile location. By careful sampling of this data structure in a similar way that the tiled resource is intended to be sampled, one might discover what the minimum LOD that is fully mapped for an entire texture filter footprint will be. To help make this process easier, Direct3D 11.2 introduces a new general purpose sampler mode, min/max filtering.

The utility of min/max filtering for LOD tracking might be useful for other purposes, such as, perhaps the filtering of depth surfaces.

Min/max reduction filtering is a mode on samplers that fetches the same set of texels that a normal texture filter would fetch. But instead of blending the values to produce an answer, it returns the `min()` or `max()` of the texels fetched, on a per-component basis (for example, the `min` of all the R values, separately from the `min` of all the G values and so on).

The min/max operations follow Direct3D arithmetic precision rules. The order of comparisons doesn't matter.

During filter operations that aren't min/max, sometimes the weight of a given texel ends up being 0.0. An example is a linear sample with texture coordinates that fall directly on a texel center - 3 other texels (which ones they are may vary by hardware) contribute to the filter but with 0 weight. For any of these texels that would be 0 weight on a non-min/max filter, if the filter is min/max, these texels still do not contribute to the result (and the weights do not otherwise affect the min/max filter operation).

The full list of filter modes is shown in the [D3D11_FILTER](#) enumeration with MINIMUM and MAXIMUM in the enumeration values.

Support for this feature depends on [Tier 2](#) support for tiled resources.

Related topics

[Pipeline access to tiled resources](#)

HLSL tiled resources exposure

11/2/2020 • 2 minutes to read • [Edit Online](#)

New Microsoft High Level Shader Language (HLSL) syntax is required to support tiled resources in [Shader Model 5](#).

The new HLSL syntax is allowed only on devices with tiled resources support. Each relevant HLSL method for tiled resources in the following table accepts either one (feedback) or two (clamp and feedback in this order) additional optional parameters. For example, a **Sample** method is:

```
Sample(sampler, location [, offset [, clamp [, feedback] ] ])
```

An example of a **Sample** method is [Texture2D.Sample\(S,float,int,float,uint\)](#).

The offset, clamp and feedback parameters are optional. You must specify all optional parameters up to the one you need, which is consistent with the C++ rules for default function arguments. For example, if the feedback status is needed, both offset and clamp parameters need to be explicitly supplied to **Sample**, even though they may not be logically needed.

The clamp parameter is a scalar float value. The literal value of clamp=0.0f indicates that the clamp operation isn't performed.

The feedback parameter is a **uint** variable that you can supply to the memory-access querying intrinsic [CheckAccessFullyMapped](#) function. You must not modify or interpret the value of the feedback parameter; but, the compiler doesn't provide any advanced analysis and diagnostics to detect whether you modified the value.

Here is the syntax of [CheckAccessFullyMapped](#):

```
bool CheckAccessFullyMapped(in uint FeedbackVar);
```

[CheckAccessFullyMapped](#) interprets the value of *FeedbackVar* and returns true if all data being accessed was mapped in the resource; otherwise, [CheckAccessFullyMapped](#) returns false.

If either the clamp or feedback parameter is present, the compiler emits a variant of the basic instruction. For example, sample of a tiled resource generates the `sample_c1_s` instruction. If neither clamp nor feedback is specified, the compiler emits the basic instruction, so that there is no change from the current behavior. The clamp value of 0.0f indicates that no clamp is performed; thus, the driver compiler can further tailor the instruction to the target hardware. If feedback is a NULL register in an instruction, the feedback is unused; thus, the driver compiler can further tailor the instruction to the target architecture.

If the HLSL compiler infers that clamp is 0.0f and feedback is unused, the compiler emits the corresponding basic instruction (for example, `sample` rather than `sample_c1_s`).

If a tiled resource access consists of several constituent byte code instructions, for example, for structured resources, the compiler aggregates individual feedback values via the OR operation to produce the final feedback value. Therefore, you see a single feedback value for such a complex access.

This is the summary table of HLSL methods that are changed to support feedback and/or clamp. These all work on tiled and non-tiled resources of all dimensions. Non-tiled resources always appear to be fully mapped.

HLSL OBJECTS	INTRINSIC METHODS WITH FEEDBACK OPTION (*) - ALSO HAS CLAMP OPTION
[RW]Texture2D [RW]Texture2DArray TextureCUBE TextureCUBEArray	Gather GatherRed GatherGreen GatherBlue GatherAlpha GatherCmp GatherCmpRed GatherCmpGreen GatherCmpBlue GatherCmpAlpha
[RW]Texture1D [RW]Texture1DArray [RW]Texture2D [RW]Texture2DArray [RW]Texture3D TextureCUBE TextureCUBEArray	Sample* SampleBias* SampleCmp* SampleCmpLevelZero SampleGrad* SampleLevel
[RW]Texture1D [RW]Texture1DArray [RW]Texture2D Texture2DMS [RW]Texture2DArray Texture2DArrayMS [RW]Texture3D [RW]Buffer [RW]ByteAddressBuffer [RW]StructuredBuffer	Load

Related topics

[Pipeline access to tiled resources](#)

Tiled resources features tiers

2/22/2020 • 2 minutes to read • [Edit Online](#)

Direct3D 11.2 exposes tiled resources support in two tiers with the [D3D11_TILED_RESOURCES_TIER](#) values.

To query whether the hardware and driver support tiled resources and at what tier level, pass the [D3D11_FEATURE_D3D11_OPTIONS1](#) value to the *Feature* parameter of [ID3D11Device::CheckFeatureSupport](#). Also, pass a pointer to the [D3D11_FEATURE_DATA_D3D11_OPTIONS1](#) structure to the *pFeatureSupportData* parameter, and pass the size of the [D3D11_FEATURE_DATA_D3D11_OPTIONS1](#) structure to the *FeatureSupportDataSize* parameter. [CheckFeatureSupport](#) returns the tier level as a [D3D11_TILED_RESOURCES_TIER](#) value in the [TiledResourcesTier](#) member of [D3D11_FEATURE_DATA_D3D11_OPTIONS1](#).

This section describes these two tiers.

In this section

TOPIC	DESCRIPTION
Tier 1	This section describes tier 1 support.
Tier 2	This section describes tier 2 support.

Related topics

[Tiled resources](#)

Tier 1

11/2/2020 • 2 minutes to read • [Edit Online](#)

This section describes tier 1 support.

- Hardware at feature level 11.0 minimum.
- No quilting support.
- No Texture1D or Texture3D support.
- No 2, 8 or 16 sample multisample antialiasing (MSAA) support. Only 4x is required, except no 128 bpp formats.
- No standard swizzle pattern (layout within 64KB tiles and tail mip packing is up to the hardware vendor).
- Limitations on how tiles can be accessed when there are duplicate mappings, described in [Tile access limitations with duplicate mappings](#).

Limitations affecting tier 1 only

- Tiled resources can have NULL mappings but reading from them or writing to them produces undefined results, including device removed. Applications can get around this by mapping a single dummy page to all the empty areas. Take care if you write and render to a page that is mapped to multiple render target locations because the order of writes will be undefined.
- Shader instructions for clamping LOD and mapped status feedback are not available. For more info, see [HLSL tiled resources exposure](#).
- Alignment constraints for standard tile shapes: It is only guaranteed that mips (starting from the finest) whose dimensions are all multiples of the standard tile size support the standard tile shapes and can have individual tiles arbitrarily mapped/unmapped. The first mipmap in a tiled resource that has any dimension not a multiple of standard tile size, along with all coarser mipmaps, can have a non-standard tiling shape, fitting into N 64KB tiles for this set of mips at once (N reported to the application). These N tiles are considered packed as one unit, which must be either fully mapped or fully unmapped by the application at any given time, though the mappings of each of the N tiles can be at arbitrarily disjoint locations in a tile pool.
- Tiled resources with any mipmaps not a multiple of standard tile size in all dimensions are not allowed to have an array size larger than 1.
- In order to switch between referencing tiles in a tile pool via a [Buffer](#) resource to referencing the same tiles via a [Texture](#) resource, or vice-versa, the most recent call to [UpdateTileMappings](#) or [CopyTileMappings](#) that defines mappings to those tile pool tiles must be for the same resource dimension (Buffer versus Texture*) as the resource dimension that will be used to access the tiles. Otherwise, behavior is undefined including the chance of device reset. So, for example, calling [UpdateTileMappings](#) to define tile mappings for a Buffer, then [UpdateTileMappings](#) to the same tiles in the tile pool via a [Texture2D](#) resource, then accessing the tiles via the Buffer is invalid. Work-around operations are to either redefine tile mappings for a resource when switching between Buffer and Texture (or vice versa) sharing tiles or just never sharing tiles in a tile pool between Buffer resources and Texture resources.
- Min/Max reduction filtering is not supported. For info about Min/Max reduction filtering, see [Tiled resources texture sampling features](#).

Related topics

[Tiled resources features tiers](#)

Tier 2

11/2/2020 • 2 minutes to read • [Edit Online](#)

This section describes tier 2 support.

- Hardware at Feature Level 11.1 minimum.
- All features of the previous tier (without [Tier 1](#) specific limitations) plus the additions in these following items:
 - Shader instructions for clamping LOD and mapped status feedback are available. For more info, see [HLSL tiled resources exposure](#).
 - Reads from non-mapped tiles return 0 in all non-missing components of the format, and the default for missing components.
 - Writes to non-mapped tiles are stopped from going to memory but might end up in caches that subsequent reads to the same address might or might not pick up.
 - Texture filtering with a footprint that straddles **NULL** and non-**NULL** tiles contributes 0 (with defaults for missing format components) for texels on **NULL** tiles into the overall filter operation. Some early hardware don't meet this requirement and returns 0 (with defaults for missing format components) for the full filter result if any texels (with nonzero weight) fall on a **NULL** tile. No other hardware will be allowed to miss the requirement to include all (nonzero weighted) texels in the filter operation.
- **NULL** texel accesses cause the [CheckAccessFullyMapped](#) operation on the status feedback for a texture read to return false. This is regardless of how the texture access result might get write masked in the shader and how many components are in the texture format (the combination of which might make it appear that the texture does not need to be accessed).
- Alignment constraints for standard tile shapes: Mipmaps that fill at least one standard tile in all dimensions are guaranteed to use the standard tiling, with the remainder considered packed as a **unit** into N tiles (N reported to the application). The application can map the N tiles into arbitrarily disjoint locations in a tile pool, but must either map all or none of the packed tiles. The mip packing is a unique set of packed tiles per array slice.
- Min/Max reduction filtering is supported. For info about Min/Max reduction filtering, see [Tiled resources texture sampling features](#).
- Tiled resources with any mipmaps less than standard tile size in any dimension are not allowed to have an array size larger than 1.
- Limitations on how tiles can be accessed when there are duplicate mappings, described in [Tile access limitations with duplicate mappings](#), continue to apply.

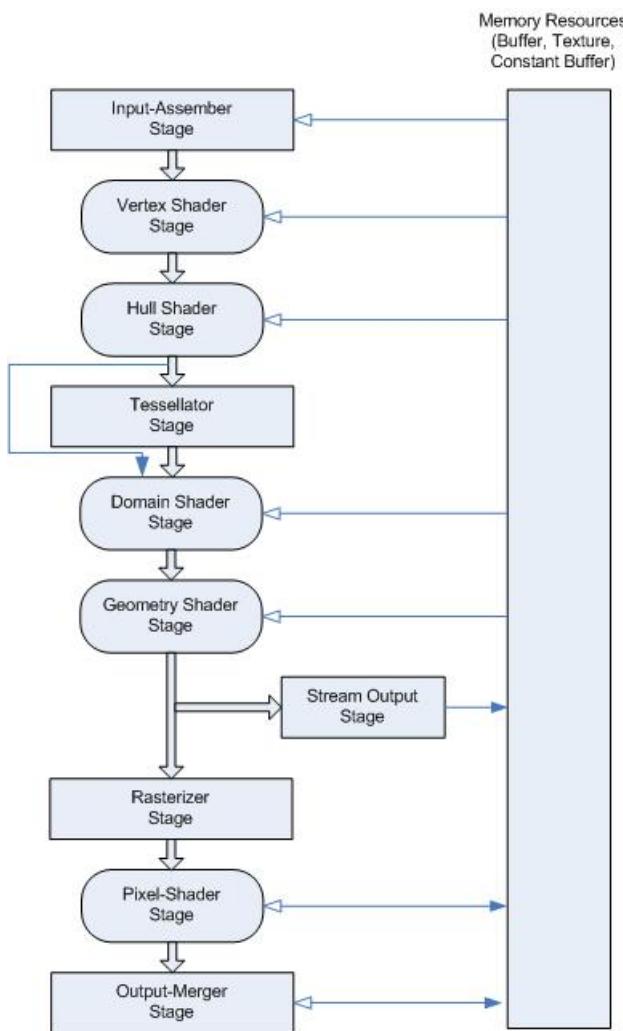
Related topics

[Tiled resources features tiers](#)

Graphics Pipeline

2/4/2021 • 2 minutes to read • [Edit Online](#)

The Direct3D 11 programmable pipeline is designed for generating graphics for realtime gaming applications. This section describes the Direct3D 11 programmable pipeline. The following diagram shows the data flow from input to output through each of the programmable stages.



The graphics pipeline for Microsoft Direct3D 11 supports the same stages as the [Direct3D 10 graphics pipeline](#), with additional stages to support advanced features.

You can use the Direct3D 11 API to configure all of the stages. Stages that feature common shader cores (the rounded rectangular blocks) are programmable by using the [HLSL](#) programming language. As you will see, this makes the pipeline extremely flexible and adaptable.

In this section

TOPIC	DESCRIPTION
Input-Assembler Stage	The Direct3D 10 and higher API separates functional areas of the pipeline into stages; the first stage in the pipeline is the input-assembler (IA) stage.

Topic	Description
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.
Tessellation Stages	The Direct3D 11 runtime supports three new stages that implement tessellation, which converts low-detail subdivision surfaces into higher-detail primitives on the GPU. Tessellation tiles (or breaks up) high-order surfaces into suitable structures for rendering.
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.
Stream-Output Stage	The purpose of the stream-output stage is to continuously output (or stream) vertex data from the geometry-shader stage (or the vertex-shader stage if the geometry-shader stage is inactive) to one or more buffers in memory (see Getting Started with the Stream-Output Stage).
Rasterizer Stage	The rasterization stage converts vector information (composed of shapes or primitives) into a raster image (composed of pixels) for the purpose of displaying real-time 3D graphics.
Pixel Shader Stage	The pixel-shader stage (PS) enables rich shading techniques such as per-pixel lighting and post-processing. A pixel shader is a program that combines constant variables, texture data, interpolated per-vertex values, and other data to produce per-pixel outputs. The rasterizer stage invokes a pixel shader once for each pixel covered by a primitive, however, it is possible to specify a NULL shader to avoid running a shader.
Output-Merger Stage	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.

Related topics

[Compute Shader](#)

[Programming Guide for Direct3D 11](#)

Input-Assembler Stage

2/4/2021 • 2 minutes to read • [Edit Online](#)

The Direct3D 10 and higher 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.

Adjacency information is visible to an application only in a geometry shader. If a geometry shader were invoked with a triangle including adjacency, for instance, the input data would contain 3 vertices for each triangle and 3 vertices for adjacency data per triangle.

When the input-assembler stage is requested to output adjacency data, the input data must include adjacency data. This may require providing a dummy vertex (forming a degenerate triangle), or perhaps by flagging in one of the vertex attributes whether the vertex exists or not. This would also need to be detected and handled by a geometry shader, although culling of degenerate geometry will happen in the rasterizer stage.

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](#).

In this section

TOPIC	DESCRIPTION
Getting Started with the Input-Assembler Stage	There are a few steps necessary to initialize the input-assembler (IA) 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.
Primitive Topologies	Direct3D 10 and higher supports several primitive types (or topologies) that are represented by the D3D_PRIMITIVE_TOPOLOGY enumerated type. These types define how vertices are interpreted and rendered by the pipeline.
Using the Input-Assembler Stage without Buffers	Creating and binding buffers is not necessary if your shaders don't require buffers. This section contains an example of simple vertex and pixel shaders that draw a single triangle.

TOPIC	DESCRIPTION
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.

Related topics

[Graphics Pipeline](#)

[Pipeline Stages \(Direct3D 10\)](#)

Getting Started with the Input-Assembler Stage

11/2/2020 • 6 minutes to read • [Edit Online](#)

There are a few steps necessary to initialize the input-assembler (IA) 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.

The basic steps involved in setting up the IA stage, shown in the following table, are covered in this topic.

STEP	DESCRIPTION
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 by using an input-layout object.
Bind 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.
Call Draw Methods	Send the data bound to the IA stage through the pipeline.

After you understand these steps, move on to [Using System-Generated Values](#).

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.

After you create the buffer resources, 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 don't use buffers. For an example of a simple vertex and pixel shader that draws a single triangle, see [Using the Input-Assembler Stage without Buffers](#).

For help with creating a vertex buffer, see [Create a Vertex Buffer](#). For help with creating an index buffer, see [Create an Index Buffer](#).

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 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 resources.

An input-layout object is created from an array of input-element descriptions and a pointer to the compiled shader (see [ID3D11Device::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 set of input-element descriptions describes all of the vertex-data elements from all of the vertex buffers that will be bound to the IA stage.

The following layout description describes a single vertex buffer that contains three vertex-data elements:

```
D3D11_INPUT_ELEMENT_DESC layout[] =
{
    { L"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
        D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { L"TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 12,
        D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { L"NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 20,
        D3D11_INPUT_PER_VERTEX_DATA, 0 },
};
```

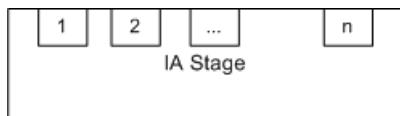
An input-element description describes each element contained by a single vertex in a vertex buffer, including size, type, location, and purpose. Each row identifies the type of data by using the semantic, the semantic index, and the data format. A [semantic](#) is a text string that identifies how the 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 normal data.

In this example of an input-element 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, 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 the time of resource creation, or you may create a resource by using a [DXGI_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*, as shown in the following illustration. 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. 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*. 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 that there is an exact match of types and semantics. You can create a single input-layout object for many shaders, as long as all of the shader-input signatures exactly match.

Bind Objects to the Input-Assembler Stage

After you create vertex buffer resources and an input layout object, you can bind them to the IA stage by calling [ID3D11DeviceContext::IASetVertexBuffers](#) and [ID3D11DeviceContext::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 is bound; however, multiple vertex buffers can be bound by a single call to [ID3D11DeviceContext::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_pVertexBuffers, //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 is bound to the IA stage by calling [ID3D11DeviceContext::IASetIndexBuffer](#).

Specify the Primitive Type

After the input buffers have been bound, the IA stage must be told how to assemble the vertices into primitives. This is done by specifying the [primitive type](#) by calling [ID3D11DeviceContext::IASetPrimitiveTopology](#); the following code calls this function to define the data as a triangle list without adjacency:

```
g_pd3dDevice->IASetPrimitiveTopology( D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST );
```

The rest of the primitive types are listed in [D3D_PRIMITIVE_TOPOLOGY](#).

Call Draw Methods

After input resources have been bound to the pipeline, an application calls a draw method to render primitives. There are several draw methods, 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.

DRAW METHODS	DESCRIPTION
ID3D11DeviceContext::Draw	Draw non-indexed, non-instanced primitives.
ID3D11DeviceContext::DrawInstanced	Draw non-indexed, instanced primitives.

DRAW METHODS	DESCRIPTION
ID3D11DeviceContext::DrawIndexed	Draw indexed, non-instanced primitives.
ID3D11DeviceContext::DrawIndexedInstanced	Draw indexed, instanced primitives.
ID3D11DeviceContext::DrawAuto	Draw non-indexed, non-instanced primitives from input data that comes from the streaming-output stage.

Each draw method renders a single topology type. During rendering, incomplete primitives (those without enough vertices, lacking indices, partial primitives, and so on) are silently discarded.

Related topics

[Input-Assembler Stage](#)

Primitive Topologies

11/2/2020 • 2 minutes to read • [Edit Online](#)

Direct3D 10 and higher supports several primitive types (or topologies) that are represented by the [D3D_PRIMITIVE_TOPOLOGY](#) enumerated type. These types define how vertices are interpreted and rendered by the pipeline.

- [Basic Primitive Types](#)
- [Primitive Adjacency](#)
- [Winding Direction and Leading Vertex Positions](#)
- [Generating Multiple Strips](#)
- [Related topics](#)

Basic Primitive Types

The following basic primitive types are supported:

- [Point List](#)
- [Line List](#)
- [Line Strip](#)
- [Triangle List](#)
- [Triangle Strip](#)

For a visualization of each primitive type, see the diagram later in this topic in [Winding Direction and Leading Vertex Positions](#).

The input-assembler stage reads data from vertex and index buffers, assembles the data into these primitives, and then sends the data to the remaining pipeline stages. (You can use the [ID3D11DeviceContext::IASetPrimitiveTopology](#) method to specify the primitive type for the input-assembler stage.)

Primitive Adjacency

All Direct3D 10 and higher primitive types (except the point list) are available in two versions: one primitive type with adjacency and one primitive type without adjacency. Primitives with adjacency contain some of the surrounding vertices, while primitives without adjacency contain only the vertices of the target primitive. For example, the line list primitive (represented by the [D3D_PRIMITIVE_TOPOLOGY_LINELIST](#) value) has a corresponding line list primitive that includes adjacency (represented by the [D3D_PRIMITIVE_TOPOLOGY_LINELIST_ADJ](#) value.)

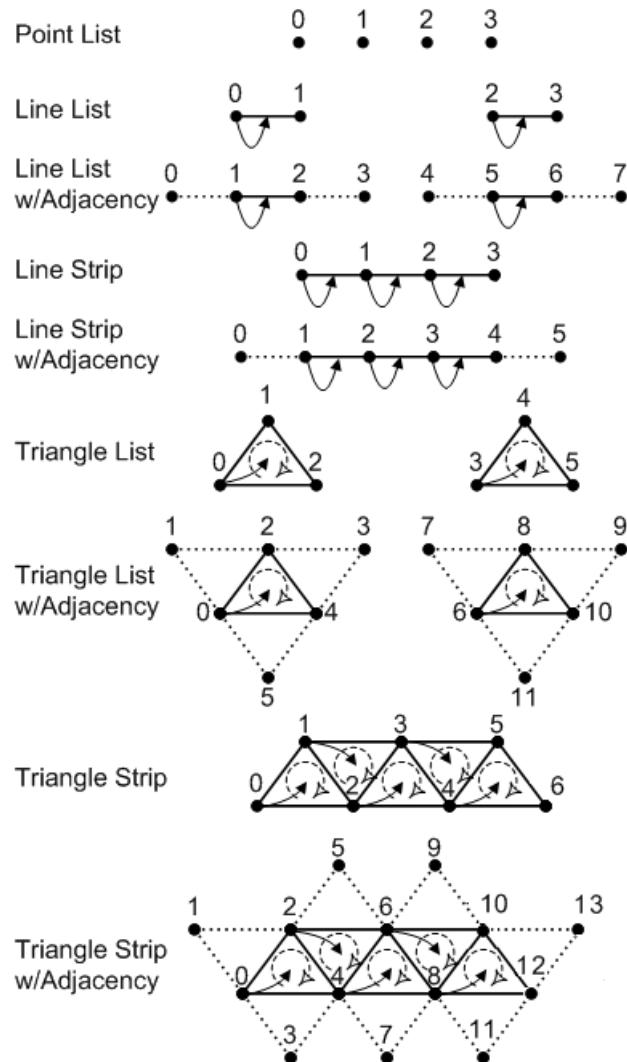
Adjacent primitives are intended to provide more information about your geometry and are only visible through a geometry shader. Adjacency is useful for geometry shaders that use silhouette detection, shadow volume extrusion, and so on.

For example, suppose you want to draw a triangle list with adjacency. A triangle list that contains 36 vertices (with adjacency) will yield 6 completed primitives. Primitives with adjacency (except line strips) contain exactly twice as many vertices as the equivalent primitive without adjacency, where each additional vertex is an adjacent vertex.

Winding Direction and Leading Vertex Positions

As shown in the following illustration, a leading vertex is the first non-adjacent vertex in a primitive. A primitive type can have multiple leading vertices defined, as long as each one is used for a different primitive. For a triangle strip with adjacency, the leading vertices are 0, 2, 4, 6, and so on. For a line strip with adjacency, the leading vertices are 1, 2, 3, and so on. An adjacent primitive, on the other hand, has no leading vertex.

The following illustration shows the vertex ordering for all of the primitive types that the input assembler can produce.



The symbols in the preceding illustration are described in the following table.

SYMBOL	NAME	DESCRIPTION
●	Vertex	A point in 3D space.
○ ↗	Winding Direction	The vertex order when assembling a primitive. Can be clockwise or counterclockwise; specify this by calling ID3D11Device1::CreateRasterizerState1 .
↗	Leading Vertex	The first non-adjacent vertex in a primitive that contains per-constant data.

Generating Multiple Strips

You can generate multiple strips through strip cutting. You can perform a strip cut by explicitly calling the [RestartStrip](#) HLSL function, or by inserting a special index value into the index buffer. This value is `-1`, which is `0xffffffff` for 32-bit indices or `0xffff` for 16-bit indices. An index of `-1` indicates an explicit 'cut' or 'restart' of the current strip. The previous index completes the previous primitive or strip and the next index starts a new primitive or strip. For more info about generating multiple strips, see [Geometry-Shader Stage](#).

NOTE

You need [feature level](#) 10.0 or higher hardware because not all 10level9 hardware implements this functionality.

Related topics

[Getting Started with the Input-Assembler Stage](#)

[Pipeline Stages \(Direct3D 10\)](#)

Using the Input-Assembler Stage without Buffers

2/22/2020 • 2 minutes to read • [Edit Online](#)

Creating and binding buffers is not necessary if your shaders don't require buffers. This section contains an example of simple vertex and pixel shaders that draw a single triangle.

- [Vertex Shader](#)
- [Pixel Shader](#)
- [Technique](#)
- [Application Code](#)
- [Related topics](#)

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

```

// 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

A technique is a collection of rendering passes (there must be at least one pass).

```

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

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

```

Application Code

```

m_pD3D11Device->IASetInputLayout( NULL );

m_pD3D11Device->IASetPrimitiveTopology( D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST );

ID3DX11EffectTechnique * pTech = NULL;
pTech = m_pEffect->GetTechniqueByIndex(0);
pTech->GetPassByIndex(iPass)->Apply(0);

m_pD3D11Device->Draw( 3, 0 );

```

Related topics

[Getting Started with the Input-Assembler Stage](#)

Using System-Generated Values

11/2/2020 • 3 minutes to read • [Edit Online](#)

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](#)
- [PrimitiveID](#)
- [InstanceId](#)
- [Example](#)
- [Related topics](#)

VertexID

A vertex id 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 [ID3D11DeviceContext::DrawIndexed](#) and [ID3D11DeviceContext::DrawIndexedInstanced](#), the vertex id represents the index value. If the vertex id overflows (exceeds $2^{32}-1$), it wraps to 0.

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

PrimitiveID

A primitive id 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 $2^{32}-1$), 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

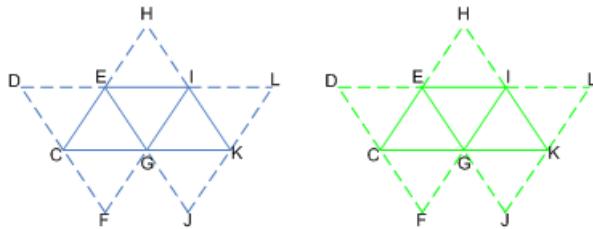
An instance id is used by each shader stage to identify the instance of the geometry that is currently being

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 $2^{32} - 1$), it wraps to 0.

Example

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



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.

PrimitiveID	0	1	2
InstanceId	0	0	0

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

PrimitiveID	0	1	2
InstanceID	1	1	1

The input assembler generates 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.

Related topics

[Input-Assembler Stage](#)

Vertex Shader Stage

11/2/2020 • 2 minutes to read • [Edit Online](#)

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.

The Vertex Shader

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: [Sample \(DirectX HLSL Texture Object\)](#), [SampleCmpLevelZero \(DirectX HLSL Texture Object\)](#), and [SampleGrad \(DirectX HLSL Texture Object\)](#)).

Related topics

[Graphics Pipeline](#)

[Pipeline Stages \(Direct3D 10\)](#)

Tessellation Stages

11/2/2020 • 8 minutes to read • [Edit Online](#)

The Direct3D 11 runtime supports three new stages that implement tessellation, which converts low-detail subdivision surfaces into higher-detail primitives on the GPU. Tessellation tiles (or breaks up) high-order surfaces into suitable structures for rendering.

By implementing tessellation in hardware, a graphics pipeline can evaluate lower detail (lower polygon count) models and render in higher detail. While software tessellation can be done, tessellation implemented by hardware can generate an incredible amount of visual detail (including support for displacement mapping) without adding the visual detail to the model sizes and paralyzing refresh rates.

- [Tessellation Benefits](#)
- [New Pipeline Stages](#)
 - [Hull-Shader Stage](#)
 - [Tessellator Stage](#)
 - [Domain-Shader Stage](#)
- [APIs for initializing Tessellation Stages](#)
- [How to's:](#)
- [Related topics](#)

Tessellation Benefits

Tessellation:

- Saves lots of memory and bandwidth, which allows an application to render higher detailed surfaces from low-resolution models. The tessellation technique implemented in the Direct3D 11 pipeline also supports displacement mapping, which can produce stunning amounts of surface detail.
- Supports scalable-rendering techniques, such as continuous or view dependent levels-of-detail which can be calculated on the fly.
- Improves performance by performing expensive computations at lower frequency (doing calculations on a lower-detail model). This could include blending calculations using blend shapes or morph targets for realistic animation or physics calculations for collision detection or soft body dynamics.

The Direct3D 11 pipeline implements tessellation in hardware, which off-loads the work from the CPU to the GPU. This can lead to very large performance improvements if an application implements large numbers of morph targets and/or more sophisticated skinning/deformation models. To access the new tessellation features, you must learn about some new pipeline stages.

New Pipeline Stages

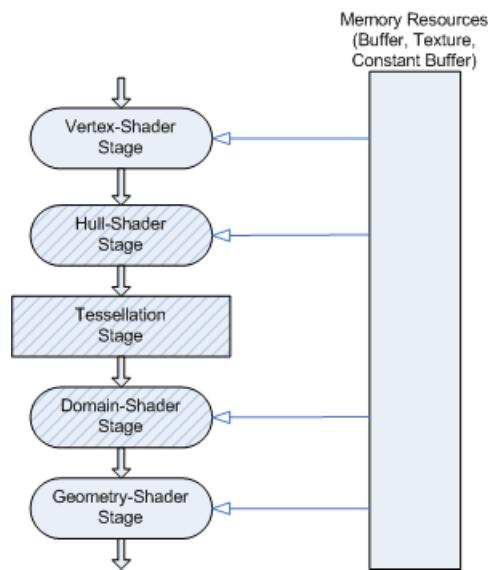
Tessellation uses the GPU to calculate a more detailed surface from a surface constructed from quad patches, triangle patches or isolines. To approximate the high-ordered surface, each patch is subdivided into triangles, points, or lines using tessellation factors. The Direct3D 11 pipeline implements tessellation using three new pipeline stages:

- [Hull-Shader Stage](#) - A programmable shader stage that produces a geometry patch (and patch constants) that correspond to each input patch (quad, triangle, or line).
- [Tessellator Stage](#) - A fixed function pipeline stage that creates a sampling pattern of the domain that represents the geometry patch and generates a set of smaller objects (triangles, points, or lines) that connect

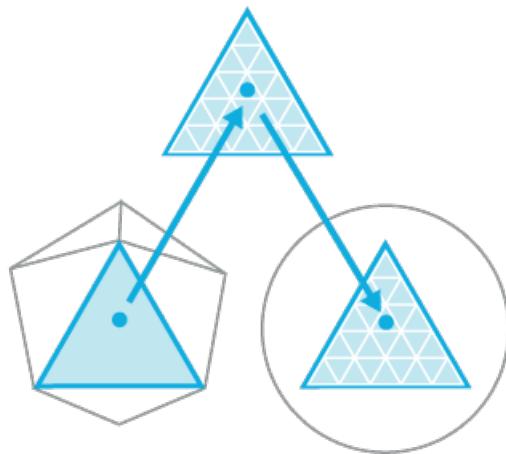
these samples.

- **Domain-Shader Stage** - A programmable shader stage that calculates the vertex position that corresponds to each domain sample.

The following diagram highlights the new stages of the Direct3D 11 pipeline.



The following diagram shows the progression through the tessellation stages. The progression starts with the low-detail subdivision surface. The progression next highlights the input patch with the corresponding geometry patch, domain samples, and triangles that connect these samples. The progression finally highlights the vertices that correspond to these samples.



Hull-Shader Stage

A hull shader -- which is invoked once per patch -- transforms input control points that define a low-order surface into control points that make up a patch. It also does some per patch calculations to provide data for the tessellation stage and the domain stage. At the simplest black-box level, the hull-shader stage would look something like the following diagram.



A hull shader is implemented with an HLSL function, and has the following properties:

- The shader input is between 1 and 32 control points.

- The shader output is between 1 and 32 control points, regardless of the number of tessellation factors. The control-points output from a hull shader can be consumed by the domain-shader stage. Patch constant data can be consumed by a domain shader; tessellation factors can be consumed by the domain shader and the tessellation stage.
- Tessellation factors determine how much to subdivide each patch.
- The shader declares the state required by the tessellator stage. This includes information such as the number of control points, the type of patch face and the type of partitioning to use when tessellating. This information appears as declarations typically at the front of the shader code.
- If the hull-shader stage sets any edge tessellation factor to = 0 or NaN, the patch will be culled. As a result, the tessellator stage may or may not run, the domain shader will not run, and no visible output will be produced for that patch.

At a deeper level, a hull-shader actually operates in two phases: a control-point phase and a patch-constant phase, which are run in parallel by the hardware. The HLSL compiler extracts the parallelism in a hull shader and encodes it into bytecode that drives the hardware.

- The control-point phase operates once for each control-point, reading the control points for a patch, and generating one output control point (identified by a ControlPointID).
- The patch-constant phase operates once per patch to generate edge tessellation factors and other per-patch constants. Internally, many patch-constant phases may run at the same time. The patch-constant phase has read-only access to all input and output control points.

Here's an example of a hull shader:

```
[patchsize(12)]
[patchconstantfunc(MyPatchConstantFunc)]
MyOutPoint main(uint Id : SV_ControlPointID,
    InputPatch<MyInPoint, 12> InPts)
{
    MyOutPoint result;

    ...

    result = TransformControlPoint( InPts[Id] );

    return result;
}
```

For an example that creates a hull shader, see [How To: Create a Hull Shader](#).

Tessellator Stage

The tessellator is a fixed-function stage initialized by binding a hull shader to the pipeline (see [How To: Initialize the Tessellator Stage](#)). The purpose of the tessellator stage is to subdivide a domain (quad, tri, or line) into many smaller objects (triangles, points or lines). The tessellator tiles a canonical domain in a normalized (zero-to-one) coordinate system. For example, a quad domain is tessellated to a unit square.

The tessellator operates once per patch using the tessellation factors (which specify how finely the domain will be tessellated) and the type of partitioning (which specifies the algorithm used to slice up a patch) that are passed in from the hull-shader stage. The tessellator outputs uv (and optionally w) coordinates and the surface topology to the domain-shader stage.

Internally, the tessellator operates in two phases:

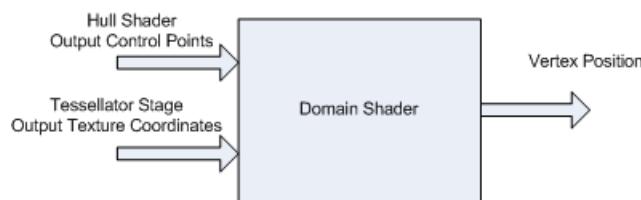
- The first phase processes the tessellation factors, fixing rounding problems, handling very small factors, reducing and combining factors, using 32-bit floating-point arithmetic.
- The second phase generates point or topology lists based on the type of partitioning selected. This is the core task of the tessellator stage and uses 16-bit fractions with fixed-point arithmetic. Fixed-point arithmetic

allows hardware acceleration while maintaining acceptable precision. For example, given a 64 meter wide patch, this precision can place points at a 2 mm resolution.

TYPE OF PARTITIONING	RANGE
fractional_odd	[1...63]
fractional_even	TessFactor range: [2..64]
integer	TessFactor range: [1..64]
pow2	TessFactor range: [1..64]

Domain-Shader Stage

A domain shader calculates the vertex position of a subdivided point in the output patch. A domain shader is run once per tessellator stage output point and has read-only access to the tessellator stage output UV coordinates, the hull shader output patch, and the hull shader output patch constants, as the following diagram shows.



Properties of the domain shader include:

- A domain shader is invoked once per output coordinate from the tessellator stage.
- A domain shader consumes output control points from the hull-shader stage.
- A domain shader outputs the position of a vertex.
- Inputs are the hull shader outputs including control points, patch constant data and tessellation factors. The tessellation factors can include the values used by the fixed function tessellator as well as the raw values (before rounding by integer tessellation, for example), which facilitates geomorphing, for example.

After the domain shader completes, tessellation is finished and pipeline data continues to the next pipeline stage (geometry shader, pixel shader etc). A geometry shader that expects primitives with adjacency (for example, 6 vertices per triangle) is not valid when tessellation is active (this results in undefined behavior, which the debug layer will complain about).

Here is an example of a domain shader:

```

void main( out MyDSOutput result,
           float2 myInputUV : SV_DomainPoint,
           MyDSInput DSInputs,
           OutputPatch<MyOutPoint, 12> ControlPts,
           MyTessFactors tessFactors)
{
    ...
    result.Position = EvaluateSurfaceUV(ControlPoints, myInputUV);
}
  
```

APIs for initializing Tessellation Stages

Tessellation is implemented with two new programmable shader stages: a hull shader and a domain shader. These new shader stages are programmed with HLSL code that is defined in shader model 5. The new shader

targets are: hs_5_0 and ds_5_0. Like all programmable shader stages, code for the hardware is extracted from compiled shaders passed into the runtime when shaders are bound to the pipeline using APIs such as [DSSetShader](#) and [HSSetShader](#). But first, the shader must be created using APIs such as [CreateHullShader](#) and [CreateDomainShader](#).

Enable tessellation by creating a hull shader and binding it to the hull-shader stage (this automatically sets up the tessellator stage). To generate the final vertex positions from the tessellated patches, you will also need to create a domain shader and bind it to the domain-shader stage. Once tessellation is enabled, the data input to the input-assembler stage must be patch data. That is, the input assembler topology must be a patch constant topology from [D3D11_PRIMITIVE_TOPOLOGY](#) set with [IASetPrimitiveTopology](#).

To disable tessellation, set the hull shader and the domain shader to **NULL**. Neither the geometry-shader stage nor the stream-output stage can read hull-shader output-control points or patch data.

- New topologies for the input-assembler stage, which are extensions to this enumeration.

```
enum D3D11_PRIMITIVE_TOPOLOGY
```

The topology is set to the input-assembler stage using [IASetPrimitiveTopology](#)

- Of course, the new programmable shader stages require other state to be set, to bind constant buffers, samples, and shader resources to the appropriate pipeline stages. These new ID3D11Device methods are implemented for setting this state.
 - [DSGetConstantBuffers](#)
 - [DSGetSamplers](#)
 - [DSGetShader](#)
 - [DSGetShaderResources](#)
 - [DSSetConstantBuffers](#)
 - [DSSetSamplers](#)
 - [DSSetShader](#)
 - [DSSetShaderResources](#)
 - [HSGetConstantBuffers](#)
 - [HSGetShaderResources](#)
 - [HSGetSamplers](#)
 - [HSGetShader](#)
 - [HSSetConstantBuffers](#)
 - [HSSetSamplers](#)
 - [HSSetShader](#)
 - [HSSetShaderResources](#)

How to's:

The documentation also contains examples for initializing the tessellation stages.

ITEM	DESCRIPTION
How To: Create a Hull Shader	Create a hull shader.
How To: Design a Hull Shader	Design a hull shader.
How To: Initialize the Tessellator Stage	Initialize the tessellation stage.

ITEM	DESCRIPTION
How To: Create a Domain Shader	Create a domain shader.
How To: Design a Domain Shader	Create a domain shader.

Related topics

[Graphics Pipeline](#)

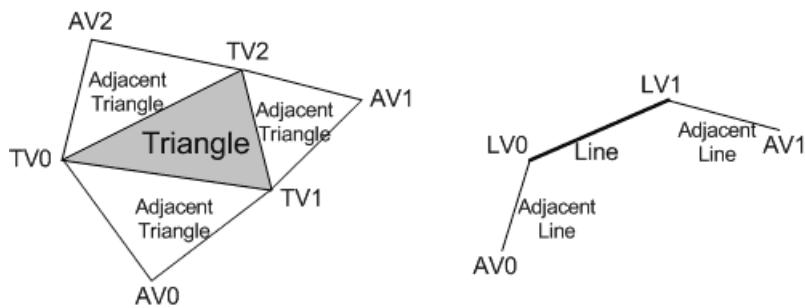
Geometry Shader Stage

11/2/2020 • 3 minutes to read • [Edit Online](#)

The geometry-shader (GS) stage runs application-specified shader code with vertices as input and the ability to generate vertices on output.

The Geometry Shader

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 following illustration shows a triangle and a line with adjacent vertices.



TV	Triangle vertex
AV	Adjacent vertex
LV	Line vertex

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 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](#)).

Related topics

[Graphics Pipeline](#)

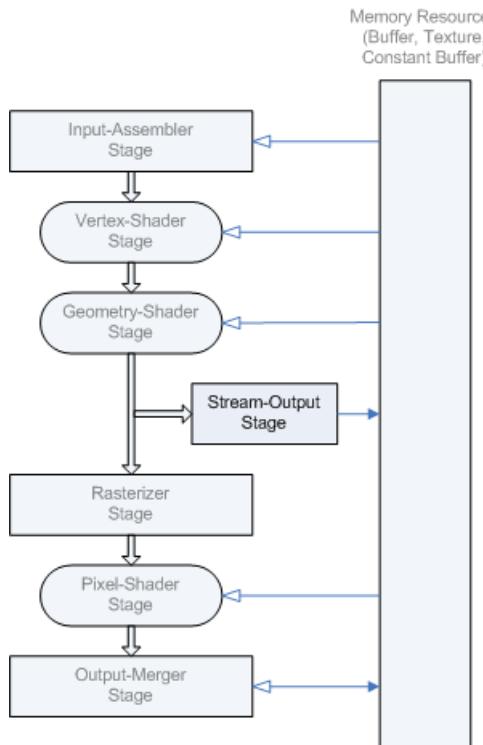
[Pipeline Stages \(Direct3D 10\)](#)

Stream-Output Stage

2/4/2021 • 2 minutes to read • [Edit Online](#)

The purpose of the stream-output stage is to continuously output (or stream) vertex data from the geometry-shader stage (or the vertex-shader stage if the geometry-shader stage is inactive) to one or more buffers in memory (see [Getting Started with the Stream-Output Stage](#)).

The stream-output stage (SO) is located in the pipeline right after the geometry-shader stage and just before the rasterization stage, as shown in the following diagram.



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 (so it can be read by the CPU). The amount of data streamed out can vary; the [ID3D11DeviceContext::DrawAuto](#) API is designed to handle the data without the need to query (the GPU) about the amount of data written.

When a triangle or line strip is bound to the input-assembler stage, each strip is converted into a list before they are streamed out. Vertices are always written out as complete primitives (for example, 3 vertices at a time for triangles); incomplete primitives are never streamed out. Primitive types with adjacency discard the adjacency data before streaming data out.

There are two ways to feed stream-output data into the pipeline:

- Stream-output data can be fed back into the input-assembler stage.
- Stream-output data can be read by programmable shaders using load functions (such as [Load](#)).

To use a buffer as a stream-output resource, create the buffer with the [D3D11_BIND_STREAM_OUTPUT](#) flag. The stream-output stage supports up to 4 buffers simultaneously.

- If you are streaming data into multiple buffers, each buffer can only capture a single element (up to 4 components) of per-vertex data, with an implied data stride equal to the element width in each buffer (compatible with the way single element buffers can be bound for input into shader stages). Furthermore, if the buffers have different sizes, writing stops as soon as any one of the buffers is full.

- If you are streaming data into a single buffer, the buffer can capture up to 64 scalar components of per-vertex data (256 bytes or less) or the vertex stride can be up to 2048 bytes.

In this section

TOPIC	DESCRIPTION
Getting Started with the Stream-Output Stage	This section describes how to use a geometry shader with the stream output stage.

Related topics

[Graphics Pipeline](#)

[Pipeline Stages \(Direct3D 10\)](#)

Getting Started with the Stream-Output Stage

11/2/2020 • 6 minutes to read • [Edit Online](#)

This section describes how to use a geometry shader with the stream output stage.

Compile a Geometry Shader

This geometry shader (GS) calculates a face normal for each triangle, and outputs position, normal and texture coordinate data.

```
struct GSPOS_INPUT
{
    float4 Pos : SV_POSITION;
    float3 Norm : TEXCOORD0;
    float2 Tex : TEXCOORD1;
};

[maxvertexcount(12)]
void GS( triangle GSPOS_INPUT input[3], inout TriangleStream<GSPOS_INPUT> TriStream )
{
    GSPOS_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) );
    float3 ExplodeAmt = faceNormal*Explode;

    //
    // Calculate the face center
    //
    float3 centerPos = (input[0].Pos.xyz + input[1].Pos.xyz + input[2].Pos.xyz)/3.0;
    float2 centerTex = (input[0].Tex + input[1].Tex + input[2].Tex)/3.0;
    centerPos += faceNormal*Explode;

    //
    // Output the pyramid
    //
    for( int i=0; i<3; i++ )
    {
        output.Pos = input[i].Pos + float4(ExplodeAmt,0);
        output.Pos = mul( output.Pos, View );
        output.Pos = mul( output.Pos, Projection );
        output.Norm = input[i].Norm;
        output.Tex = input[i].Tex;
        TriStream.Append( output );

        int iNext = (i+1)%3;
        output.Pos = input[iNext].Pos + float4(ExplodeAmt,0);
        output.Pos = mul( output.Pos, View );
        output.Pos = mul( output.Pos, Projection );
        output.Norm = input[iNext].Norm;
        output.Tex = input[iNext].Tex;
        TriStream.Append( output );

        output.Pos = float4(centerPos,1) + float4(ExplodeAmt,0);
        output.Pos = mul( output.Pos, View );
        output.Pos = mul( output.Pos, Projection );
        output.Norm = faceNormal;
```

```

        output.Tex = centerTex;
        TriStream.Append( output );

        TriStream.RestartStrip();
    }

    for( int i=2; i>=0; i-- )
    {
        output.Pos = input[i].Pos + float4(ExplodeAmt,0);
        output.Pos = mul( output.Pos, View );
        output.Pos = mul( output.Pos, Projection );
        output.Norm = -input[i].Norm;
        output.Tex = input[i].Tex;
        TriStream.Append( output );
    }
    TriStream.RestartStrip();
}

```

Keeping that code in mind, consider that a geometry shader looks much like a vertex or pixel shader, but with the following exceptions: the type returned by the function, the input parameter declarations, and the intrinsic function.

ITEM	DESCRIPTION
Function return type	<p>The function return type does one thing, declares the maximum number of vertices that can be output by the shader. In this case,</p> <div style="border: 1px solid black; padding: 10px; text-align: center;"> maxvertexcount(12) </div> <p>defines the output to be a maximum of 12 vertices.</p>
Input parameter declarations	<p>This function takes two input parameters:</p> <div style="border: 1px solid black; padding: 10px; text-align: center;"> triangle GSPS_INPUT input[3] , inout TriangleStream TriStream </div> <p>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).</p> <p>The second parameter is a triangle stream defined by the type TriangleStream. 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).</p> <p>Use the triangle and trianglestream keywords to identify individual triangles or a stream of triangles in a GS.</p>

ITEM	DESCRIPTION
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 positional 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 [D3DCompile](#) to compile as shown in the following code example.

```
DWORD dwShaderFlags = D3DCOMPILE_ENABLE_STRICTNESS;
ID3DBlob** ppShader;

D3DCompile( pSrcData, sizeof( pSrcData ),
    "Tutorial13.fx", NULL, NULL, "GS", "gs_4_0",
    dwShaderFlags, 0, &ppShader, NULL );
```

Just like vertex and pixel shaders, you need a shader flag to tell the compiler how you want the shader compiled (for debugging, optimized for speed, and so on), 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 [ID3D11Device::CreateGeometryShaderWithStreamOutput](#) to create the geometry shader object.

But first, you need to declare the stream output (SO) stage input signature. This signature matches or validates the GS outputs and the SO inputs at the time of object creation. The following code is an example of the SO declaration.

```
D3D11_SO_DECLARATION_ENTRY pDecl[] =
{
    // semantic name, semantic index, start component, component count, output slot
    { "SV_POSITION", 0, 0, 4, 0 },    // output all components of position
    { "TEXCOORD0", 0, 0, 3, 0 },     // output the first 3 of the normal
    { "TEXCOORD1", 0, 0, 2, 0 }      // output the first 2 texture coordinates
};

D3D11Device->CreateGeometryShaderWithStreamOut( pShaderBytecode, ShaderBytecodesize, pDecl,
    sizeof(pDecl), NULL, 0, 0, NULL, &pStreamOutGS );
```

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 vertex shader). For information about how 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 [D3D11_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 that is created (see [ID3D11GeometryShader](#)).

In this situation, the buffer stride is NULL, the index of the stream to be sent to the rasterizer is 0, and the class linkage interface is NULL.

The stream output declaration defines the way that 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

[ID3D11Device::CreateGeometryShaderWithStreamOutput](#) and pass a pointer to a vertex shader to the *pShaderBytecode* parameter.

Set the Output Targets

The last step is to set the SO stage buffers. Data can be streamed into one or more buffers in memory for use later. The following code 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.

```
ID3D11Buffer *m_pBuffer;
int m_nBufferSize = 1000000;

D3D11_BUFFER_DESC bufferDesc =
{
    m_nBufferSize,
    D3D11_USAGE_DEFAULT,
    D3D11_BIND_STREAM_OUTPUT,
    0,
    0,
    0
};
D3D11Device->CreateBuffer( &bufferDesc, NULL, &m_pBuffer );
```

Create a buffer by calling [ID3D11Device::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 bind flag D3D10_BIND_STREAM_OUTPUT.

Once the buffer is successfully created, set it to the current device by calling

[ID3D11DeviceContext::SOSetTargets](#):

```
UINT offset[1] = 0;
D3D11Device->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 function is called. An internal variable keeps track of the position for where to begin writing data to the streaming-output buffers, and that variables 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.

Related topics

[Stream-Output Stage](#)

Rasterizer Stage

2/4/2021 • 2 minutes to read • [Edit Online](#)

The rasterization stage converts vector information (composed of shapes or primitives) into a raster image (composed of pixels) for the purpose of displaying real-time 3D graphics.

During rasterization, each primitive is converted into pixels, while interpolating per-vertex values across each primitive. Rasterization includes clipping vertices to the view frustum, performing a divide by z to provide perspective, mapping primitives to a 2D viewport, and determining how to invoke the pixel shader. While using a pixel shader is optional, the rasterizer stage always performs clipping, a perspective divide to transform the points into homogeneous space, and maps the vertices to the viewport.

Vertices (x,y,z,w), coming into the rasterizer stage are assumed to be in homogeneous 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 [ID3D11DeviceContext::PSSetShader](#)), and disabling depth and stencil testing (set **DepthEnable** and **StencilEnable** to **FALSE** in [D3D11_DEPTH_STENCIL_DESC](#)). While disabled, rasterization-related pipeline counters will not update. There is also a complete description of the [rasterization rules](#).

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.

In this section

TOPIC	DESCRIPTION
Getting Started with the Rasterizer Stage	This section describes setting the viewport, the scissors rectangle, the rasterizer state, and multi-sampling.
Rasterization Rules	Rasterization rules define how vector data is mapped into raster data. The raster data is snapped to integer locations that are then culled and clipped (to draw the minimum number of pixels), and per-pixel attributes are interpolated (from per-vertex attributes) before being passed to a pixel shader.

Related topics

[Graphics Pipeline](#)

[Pipeline Stages \(Direct3D 10\)](#)

Getting Started with the Rasterizer Stage

11/2/2020 • 5 minutes to read • [Edit Online](#)

This section describes setting the viewport, the scissors rectangle, the rasterizer state, and multi-sampling.

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:

```
X = (X + 1) * Viewport.Width * 0.5 + Viewport.TopLeftX  
Y = (1 - Y) * Viewport.Height * 0.5 + Viewport.TopLeftY  
Z = Viewport.MinDepth + Z * (Viewport.MaxDepth - Viewport.MinDepth)
```

Tutorial 1 creates a 640×480 viewport using [D3D11_VIEWPORT](#) and by calling [ID3D11DeviceContext::RSSetViewports](#).

```
D3D11_VIEWPORT vp[1];  
vp[0].Width = 640.0f;  
vp[0].Height = 480.0f;  
vp[0].MinDepth = 0;  
vp[0].MaxDepth = 1;  
vp[0].TopLeftX = 0;  
vp[0].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 scissor rectangles) that can be bound to the rasterizer stage at any one time is 16 (specified by [D3D11_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. The size of the scissor rectangle is specified in integers. Only one scissor rectangle (based on *ViewportArrayIndex* in [system value semantics](#)) can be applied to a triangle during rasterization.

To enable the scissor rectangle, use the *ScissorEnable* member (in [D3D11_RASTERIZER_DESC1](#)). 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 [ID3D11DeviceContext::RSSetScissorRects](#) with [D3D11_RECT](#).

```
D3D11_RECT rect[1];
rects[0].left = 0;
rects[0].right = 640;
rects[0].top = 0;
rects[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

Beginning with 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 [ID3D11Device1::CreateRasterizerState1](#) to create a rasterizer state object from a rasterizer description (see [D3D11_RASTERIZER_DESC1](#)).

```
ID3D11RasterizerState1 * g_pRasterState;

D3D11_RASTERIZER_DESC1 rasterizerState;
rasterizerState.FillMode = D3D11_FILL_SOLID;
rasterizerState.CullMode = D3D11_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;
rasterizerState.ForcedSampleCount = 0;
pd3DDevice->CreateRasterizerState1( &rasterizerState, &g_pRasterState );
```

This example set of state accomplishes perhaps the most basic rasterizer setup:

- Solid fill mode
- Cull out or 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 anti-aliasing

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);
```

Multisampling

Multisampling samples some or all of the components of an image at a higher resolution (followed by downsampling to the original resolution) to reduce the most visible form of aliasing caused by drawing polygon edges. Even though multisampling requires sub-pixel samples, modern GPU's implement multisampling so that a pixel shader runs once per pixel. This provides an acceptable tradeoff between performance (especially in a GPU bound application) and anti-aliasing the final image.

To use multisampling, set the enable field in the [rasterization description](#), create a multisampled render target, and either read the render target with a shader to resolve the samples into a single pixel color or call [ID3D11DeviceContext::ResolveSubresource](#) to resolve the samples using the video card. The most common scenario is to draw to one or more multisampled render targets.

Multisampling is independent of whether or not a sample mask is used, [alpha-to-coverage](#) is enabled, or stencil operations (which are always performed per-sample).

Depth testing is affected by multisampling:

- When multisampling is enabled, depth is interpolated per sample and the depth/stencil test is done per sample; the pixel shader output color is duplicated for all passing samples. If the pixel shader outputs depth, the depth value is duplicated for all samples (although this scenario loses the benefit of multisampling).
- When multisampling is disabled, depth/stencil testing is still done per-sample, but depth is not interpolated per-sample.

There are no restrictions for mixing multisampled and non-multisampled rendering within a single render target. If you enable multisampling and draw to a non-multisampled render target, you produce the same result as if multisampling were not enabled; sampling is done with a single sample per-pixel.

Related topics

[Rasterizer Stage](#)

Rasterization Rules

11/2/2020 • 8 minutes to read • [Edit Online](#)

Rasterization rules define how vector data is mapped into raster data. The raster data is snapped to integer locations that are then culled and clipped (to draw the minimum number of pixels), and per-pixel attributes are interpolated (from per-vertex attributes) before being passed to a pixel shader.

There are several types of rules, which depend on the type of primitive that is being mapped, as well as whether or not the data uses multisampling to reduce aliasing. The following illustrations demonstrate how the corner cases are handled.

- [Triangle Rasterization Rules \(Without Multisampling\)](#)
- [Line Rasterization Rules \(Aliased, Without Multisampling\)](#)
- [Line Rasterization Rules \(Antialiased, Without Multisampling\)](#)
- [Point Rasterization Rules \(Without Multisampling\)](#)
- [Multisample Anti-Aliasing Rasterization Rules](#)
 - [Hardware Support](#)
 - [Centroid Sampling of Attributes when Multisample Antialiasing](#)
 - [Derivative Calculations When Multisampling](#)
- [Related topics](#)

Triangle Rasterization Rules (Without Multisampling)

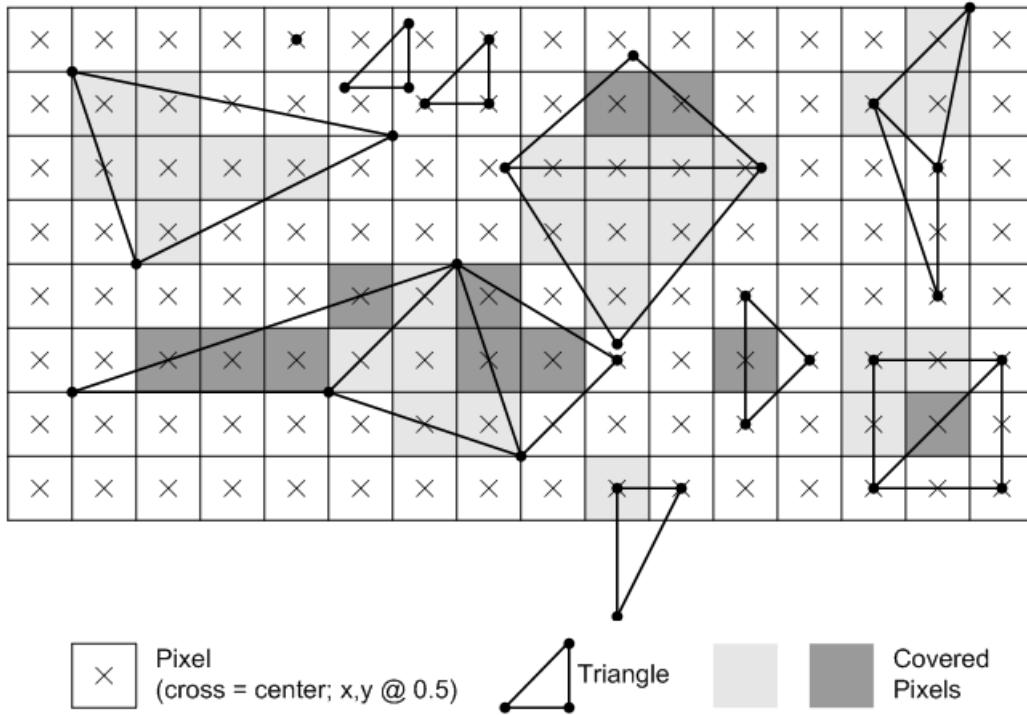
Any pixel center which falls inside a triangle is drawn; a pixel is assumed to be inside if it passes the top-left rule. The top-left rule is that a pixel center is defined to lie inside of a triangle if it lies on the top edge or the left edge of a triangle.

Where:

- A top edge, is an edge that is exactly horizontal and is above the other edges.
- A left edge, is an edge that is not exactly horizontal and is on the left side of the triangle. A triangle can have one or two left edges.

The top-left rule ensures that adjacent triangles are drawn once.

This illustration shows examples of pixels that are drawn because they either lie inside a triangle or follow the top-left rule.



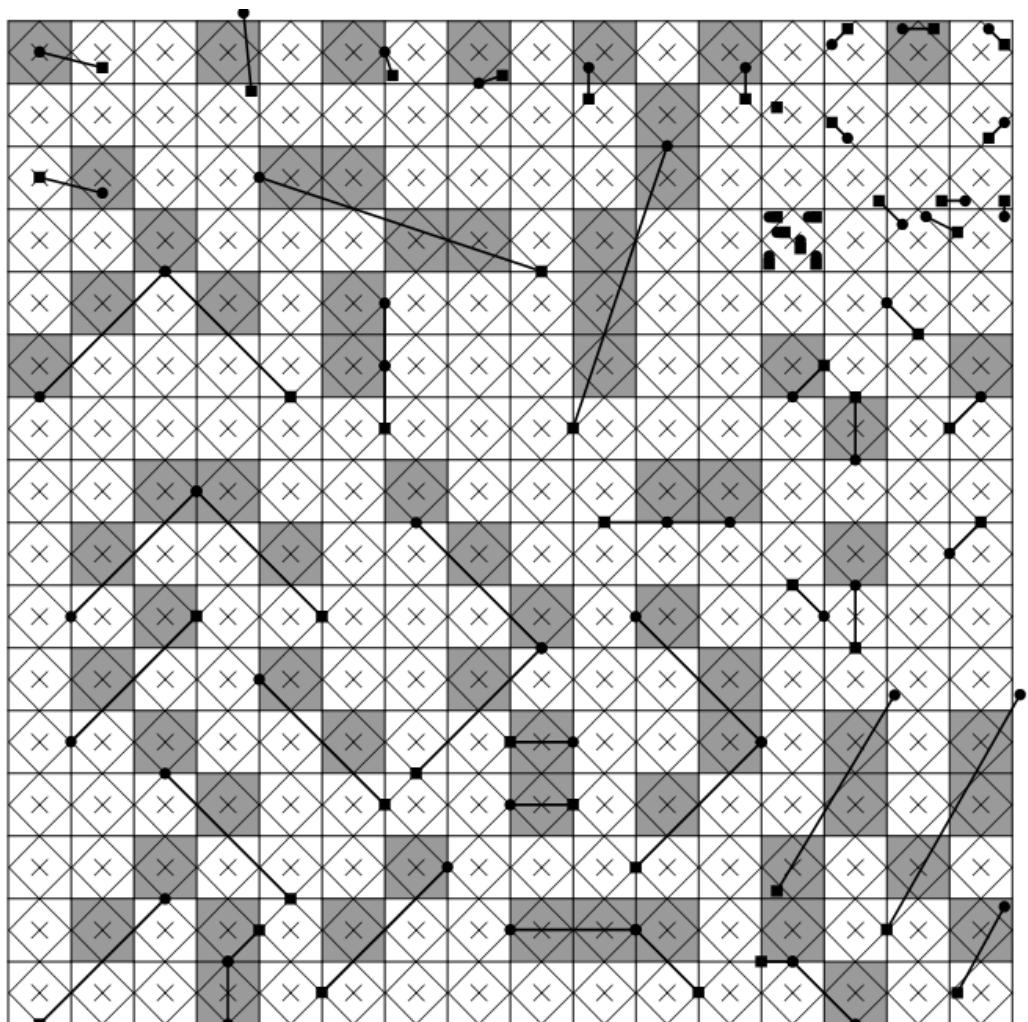
The light and dark gray covering of the pixels show them as groups of the pixels to indicate which triangle they are inside.

Line Rasterization Rules (Aliased, Without Multisampling)

Line rasterization rules use a diamond test area to determine if a line covers a pixel. For x-major lines (lines with $-1 \leq \text{slope} \leq +1$), the diamond test area includes (shown solid) the lower-left edge, lower-right edge, and bottom corner; the diamond excludes (shown dotted) the upper-left edge, upper-right edge, the top corner, the left corner, and the right corner. A y-major line is any line that is not an x-major line; the test diamond area is the same as described for the x-major line except the right corner is also included.

Given the diamond area, a line covers a pixel if the line exits the pixel's diamond test area when traveling along the line from the start towards the end. A line strip behaves the same, as it is drawn as a sequence of lines.

The following illustration shows some examples.

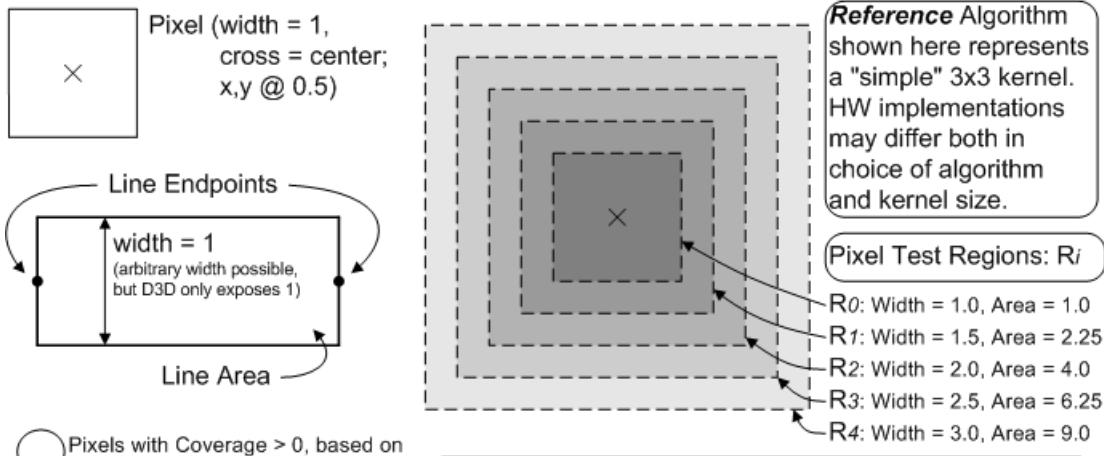


- X Pixel (cross = center; $x,y @ 0.5$)
- Pixel Covered
- Diamond Test Area when x-major ($-1 \leq \text{slope} \leq 1$)
- Diamond Test Area when y-major (all other slopes)
- Independent Line
dot = start, square = end
- Line Strip
dot = start/interior, square = end

Line Rasterization Rules (Antialiased, Without Multisampling)

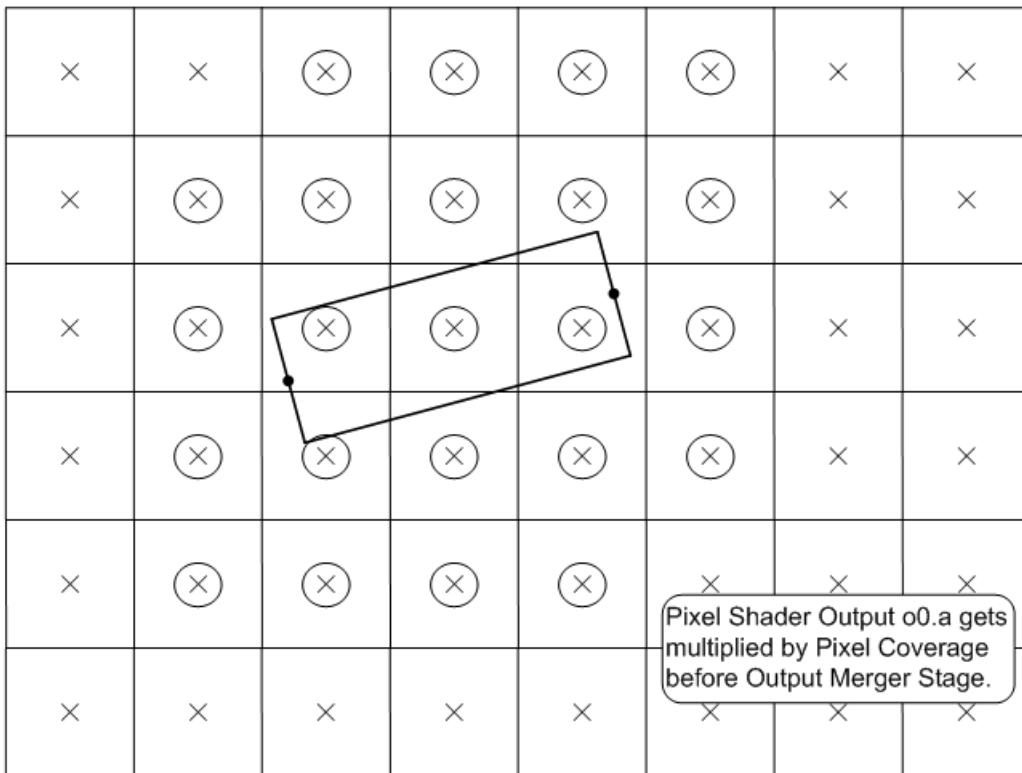
An antialiased line is rasterized as if it were a rectangle (with width = 1). The rectangle intersects with a render target producing per-pixel coverage values, which are multiplied into pixel shader output alpha components. There is no antialiasing performed when drawing lines on a multisampled render target.

It is deemed that there is no single "best" way to perform antialiased line rendering. Direct3D 10 adopts as a guideline the method shown in the following illustration. This method was derived empirically, exhibiting a number of visual properties deemed desirable. Hardware need not exactly match this algorithm; tests against this reference shall have "reasonable" tolerances, guided by some of the principles listed further below, permitting various hardware implementations and filter kernel sizes. None of this flexibility permitted in hardware implementation, however, can be communicated up through Direct3D 10 to applications, beyond simply drawing lines and observing/measuring how they look.



Pixels with Coverage > 0, based on formula to the right. Pixel Shader is invoked on these pixels (at centers), plus "silent" pixels as needed to round out 2x2 stamps for derivatives.

$$\text{Pixel Coverage} = \sum_{i: 0 \text{ to } 4} \frac{\text{AreaOfLineIntersectionWith}(R_i)}{\text{Area}(R_i) * 5}$$

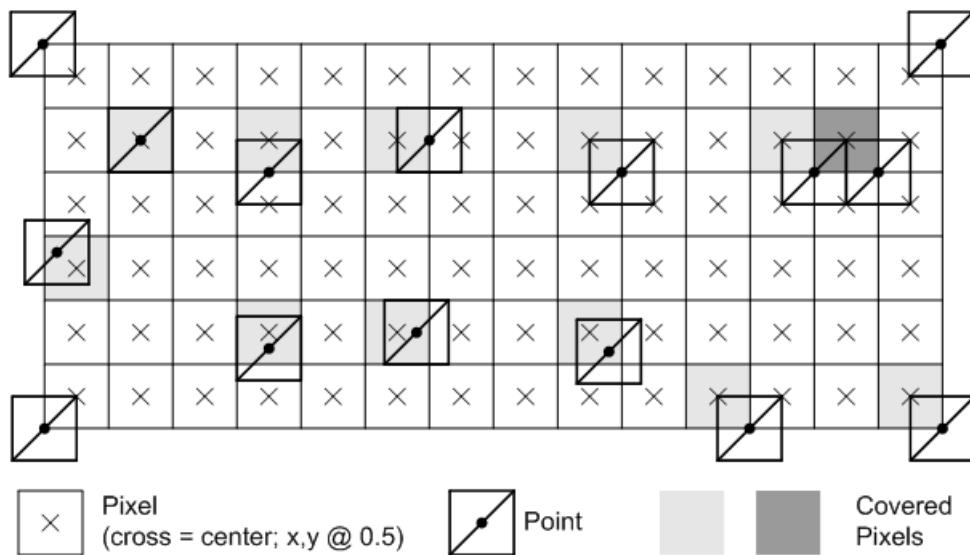


This algorithm generates relatively smooth lines, with uniform intensity, with minimal jagged edges or braiding. Moire patterning for close lines is minimized. There is good coverage for junctions between line segments placed end-to-end. The filter kernel is a reasonable tradeoff between the amount of edge blurring and the changes in intensity caused by gamma corrections. The coverage value is multiplied into pixel shader o0.a (srcAlpha) per the following formula by the output-merger stage: $\text{srcColor} * \text{srcAlpha} + \text{destColor} * (1 - \text{srcAlpha})$.

Point Rasterization Rules (Without Multisampling)

A point is interpreted as though it were composed of two triangles in a Z pattern, which use triangle rasterization rules. The coordinate identifies the center of a one pixel wide square. There is no culling for points.

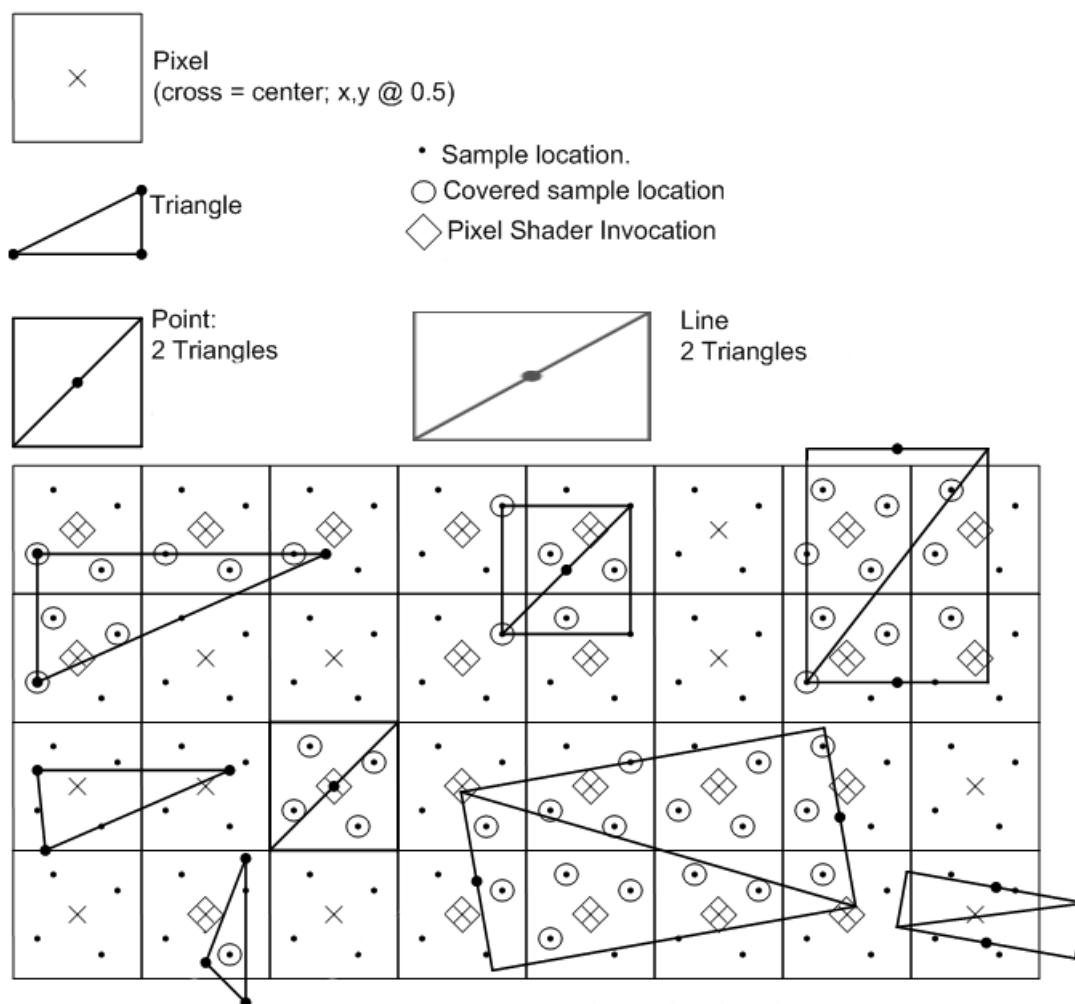
The following illustration shows some examples.



Multisample Anti-Aliasing Rasterization Rules

Multisample antialiasing (MSAA) reduces geometry aliasing using pixel coverage and depth-stencil tests at multiple sub-sample locations. To improve performance, per-pixel calculations are performed once for each covered pixel, by sharing shader outputs across covered sub-pixels. Multisample antialiasing does not reduce surface aliasing. Sample locations and reconstruction functions are dependent on the hardware implementation.

The following illustration shows some examples.



The number of sample locations is dependent on the multisample mode. Vertex attributes are interpolated at

pixel centers, since this is where the pixel shader is invoked (this becomes extrapolation if the center is not covered). Attributes can be flagged in the pixel shader to be centroid sampled, which causes non-covered pixels to interpolate the attribute at intersection of the pixel's area and the primitive. A pixel shader runs for each 2x2 pixel area to support derivative calculations (which use x and y deltas). This means that shader invocations occur more than is shown to fill out the minimum 2x2 quanta (which is independent of multisampling). The shader result is written out for each covered sample that passes the per-sample depth-stencil test.

Rasterization rules for primitives are, in general, unchanged by multisample antialiasing, except:

- For a triangle, a coverage test is performed for each sample location (not for a pixel center). If more than one sample location is covered, a pixel shader runs once with attributes interpolated at the pixel center. The result is stored (replicated) for each covered sample location in the pixel that passes the depth/stencil test.

A line is treated as a rectangle made up of two triangles, with a line width of 1.4.

- For a point, a coverage test is performed for each sample location (not for a pixel center).

Many formats support multisampling (see [Hardware Support for Direct3D 10 Formats](#)), some formats can be resolved ([ResolveSubresource](#); which downsamples a multisampled format to a sample size of 1).

Multisampling formats can be used in render targets which can be read back into shaders using [load](#), since no resolve is required for individual samples accessed by the shader. Depth formats are not supported for multisample resource, therefore, depth formats are restricted to render targets only.

Typeless formats (R8G8B8A8_TYPELESS for instance) support multisampling to allow a resource view to interpret data in different ways. For instance, you could create a multisample resource using R8G8B8A8_TYPELESS, render to it using a render-target-view resource with a R8G8B8A8_UINT format, then resolve the contents to another resource with a R8G8B8A8_UNORM data format.

Hardware Support

The API reports hardware support for multisampling through the number of quality levels. For example, a 0 quality level means the hardware does not support multisampling (at a particular format and quality level). A 3 for quality levels means that the hardware supports three different sample layouts and/or resolve algorithms. You can also assume the following:

- Any format that supports multisampling, supports the same number of quality levels for every format in that family.
- Every format that supports multisampling, and has the _UNORM, _SRGB, _SNORM or _FLOAT formats, also supports resolving.

Centroid Sampling of Attributes when Multisample Antialiasing

By default, vertex attributes are interpolated to a pixel center during multisample antialiasing; if the pixel center is not covered, attributes are extrapolated to a pixel center. If a pixel shader input that contains the centroid semantic (assuming the pixel is not fully covered) will be sampled somewhere within the covered area of the pixel, possibly at one of the covered sample locations. A sample mask (specified by the rasterizer state) is applied prior to centroid computation. Therefore, a sample that is masked out will not be used as a centroid location.

The reference rasterizer chooses a sample location for centroid sampling similar to this:

- The sample mask allows all samples. Use a pixel center if the pixel is covered or if none of the samples are covered. Otherwise, the first covered sample is chosen, starting from the pixel center and moving outward.
- The sample mask turns off all samples but one (a common scenario). An application can implement multipass supersampling by cycling through single-bit sample-mask values and re-rendering the scene for each sample using centroid sampling. This would require that an application adjust derivatives to select appropriately more detailed texture mips for the higher texture sampling density.

Derivative Calculations When Multisampling

Pixel shaders always run using a minimum 2x2 pixel area to support derivative calculations, which are calculated by taking deltas between data from adjacent pixels (making the assumption that the data in each pixel has been sampled with unit spacing horizontally or vertically). This is unaffected by multisampling.

If derivatives are requested on an attribute that has been centroid sampled, the hardware calculation is not adjusted, which can cause inaccurate derivatives. A shader will expect a unit vector in render-target space but may get a non-unit vector with respect to some other vector space. Therefore, it is an application's responsibility to exhibit caution when requesting derivatives from attributes that are centroid sampled. In fact, it is recommended that you do not combine derivatives and centroid sampling. Centroid sampling can be useful for situations where it is critical that a primitive's interpolated attributes are not extrapolated, but this comes with tradeoffs such as attributes that appear to jump where a primitive edge crosses a pixel (rather than change continuously) or derivatives that cannot be used by texture sampling operations that derive LOD.

Related topics

[Rasterizer Stage](#)

Pixel Shader Stage

11/2/2020 • 2 minutes to read • [Edit Online](#)

The pixel-shader stage (PS) enables rich shading techniques such as per-pixel lighting and post-processing. A pixel shader is a program that combines constant variables, texture data, interpolated per-vertex values, and other data to produce per-pixel outputs. The rasterizer stage invokes a pixel shader once for each pixel covered by a primitive, however, it is possible to specify a **NONE** shader to avoid running a shader.

The Pixel Shader

When multisampling a texture, a pixel shader is invoked once per-covered pixel while a depth/stencil test occurs for each covered multisample. Samples that pass the depth/stencil test are updated with the pixel shader output color.

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, a hardware implementation runs 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.

Inputs

When the pipeline is configured without a geometry shader, a pixel shader is limited to 16, 32-bit, 4-component inputs. Otherwise, a pixel shader can take up to 32, 32-bit, 4-component inputs.

Pixel shader input data includes vertex attributes (that can be interpolated with or without perspective correction) or can be treated as per-primitive constants. Pixel shader inputs are interpolated from the vertex attributes of the primitive being rasterized, based on the interpolation mode declared. If a primitive gets clipped before rasterization, the interpolation mode is honored during the clipping process as well.

Vertex attributes are interpolated (or evaluated) at pixel shader center locations. Pixel shader attribute interpolation modes are declared in an input register declaration, on a per-element basis in either an [argument](#) or an [input structure](#). Attributes can be interpolated linearly, or with [centroid sampling](#). Centroid evaluation is relevant only during multisampling to cover cases where a pixel is covered by a primitive but a pixel center may not be; centroid evaluation occurs as close as possible to the (non-covered) pixel center.

Inputs may also be declared with a [system-value semantic](#), which marks a parameter that is consumed by other pipeline stages. For instance, a pixel position should be marked with the SV_Position semantic. The IA stage can produce one scalar for a pixel shader (using SV_PrimitiveID); the rasterizer stage can also generate one scalar for a pixel shader (using SV_IsFrontFace).

Outputs

A pixel shader can output up to 8, 32-bit, 4-component colors, or no color if the pixel is discarded. Pixel shader output register components must be declared before they can be used; each register is allowed a distinct output-write mask.

Use the depth-write-enable state (in the output-merger stage) to control whether depth data gets written to a depth buffer (or use the discard instruction to discard data for that pixel). A pixel shader can also output an optional 32-bit, 1-component, floating-point, depth value for depth testing (using the SV_Depth semantic). The depth value is output in the oDepth register, and replaces the interpolated depth value for depth testing (assuming depth testing is enabled). There is no way to dynamically change between using fixed-function depth

or shader oDepth.

A pixel shader cannot output a stencil value.

Related topics

[Graphics Pipeline](#)

[Pipeline Stages \(Direct3D 10\)](#)

Output-Merger Stage

2/4/2021 • 5 minutes to read • [Edit Online](#)

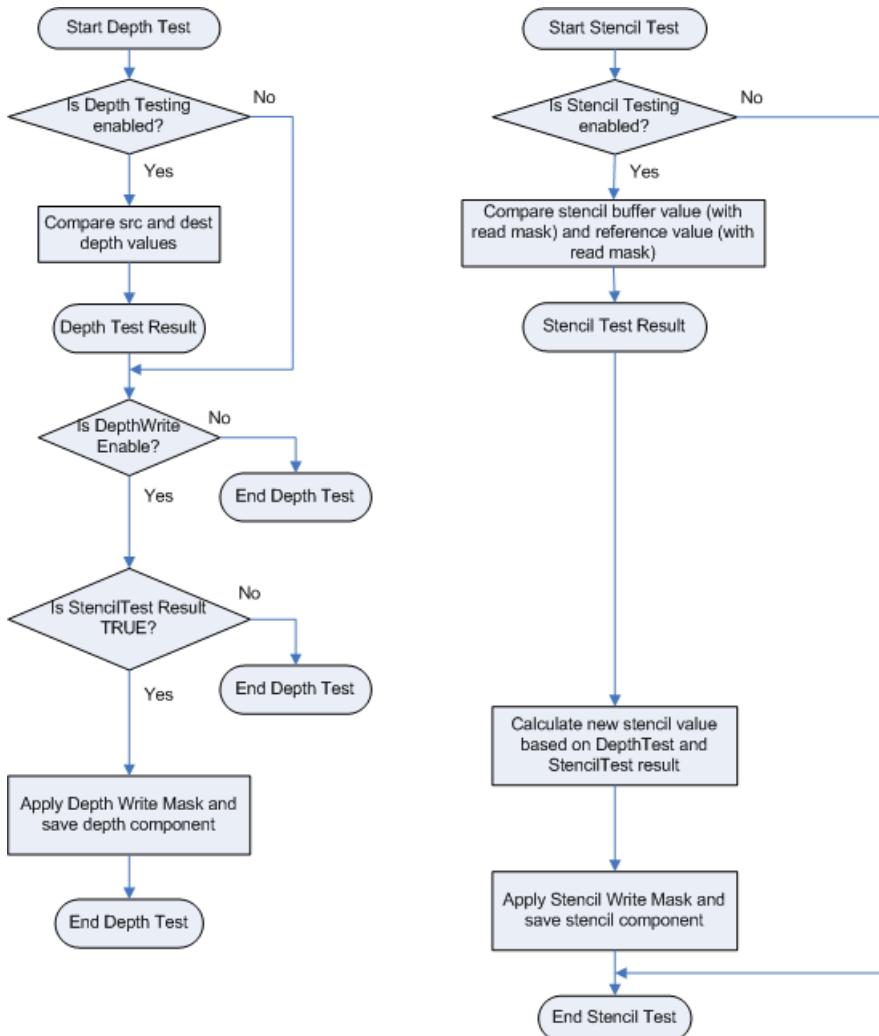
The output-merger (OM) stage generates the final rendered pixel color using a combination of pipeline state, the pixel data generated by the pixel shaders, the contents of the render targets, and the contents of the depth/stencil buffers. The OM stage is the final step for determining which pixels are visible (with depth-stencil testing) and blending the final pixel colors.

Differences between Direct3D 9 and Direct3D 10: Direct3D 9 implements alpha testing (using [alpha-testing state](#)) to control whether a pixel is written to an output render target.

Direct3D 10 and higher does not implement an alpha test (or alpha testing state). This can be controlled using a pixel shader or with depth/stencil functionality.

Depth-Stencil Testing Overview

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. The following diagram shows conceptually how depth-stencil testing is done.



To configure depth-stencil testing, see [Configuring Depth-Stencil Functionality](#). 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.

Once depth values reach the output-merger stage (whether coming from interpolation or from a pixel shader) they are always clamped: $z = \min(\text{Viewport.MaxDepth}, \max(\text{Viewport.MinDepth}, z))$ according to the format/precision of the depth buffer, using floating-point rules. After clamping, the depth value is compared (using DepthFunc) against the existing depth-buffer value. If no depth buffer is bound, the depth test always passes.

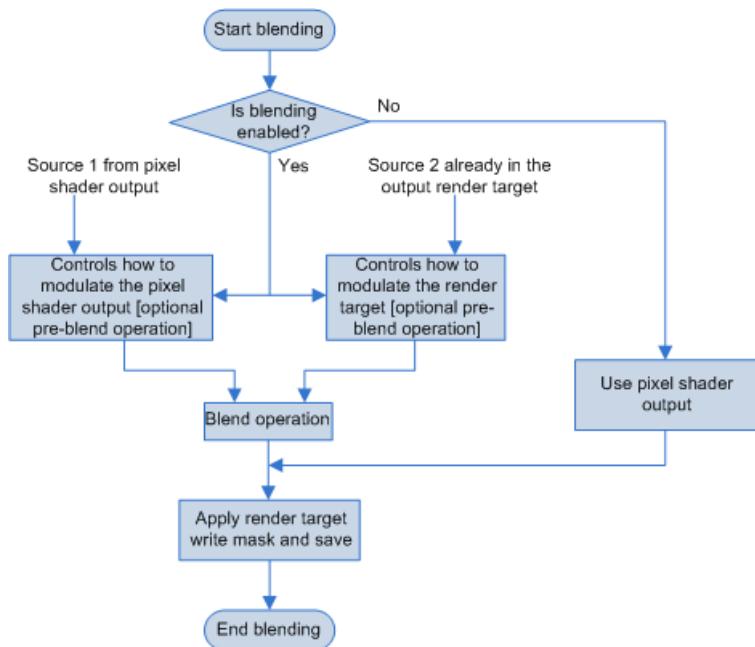
If there is no stencil component in the depth-buffer format, or no depth buffer bound, then the stencil test always passes. Otherwise, functionality is unchanged from Direct3D 9.

Only one depth/stencil buffer can be active at a time; any bound resource view must match (same size and dimensions) the depth/stencil view. This does not mean the resource size must match, just that the view size must match.

For more information about depth-stencil testing, see [tutorial 14](#).

Blending Overview

Blending combines one or more pixel values to create a final pixel color. The following diagram shows the process involved in blending pixel data.



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](#).

Fixed-function blend can be enabled independently for each render target. However there is only one set of blend controls, so that the same blend is applied to all RenderTargets with blending enabled. Blend values (including BlendFactor) are always clamped to the range of the render-target format before blending. Clamping is done per render target, respecting the render target type. The only exception is for the float16, float11 or float10 formats which are not clamped so that blend operations on these formats can be done with at least equal precision/range as the output format. NaN's and signed zeros are propagated for all cases (including 0.0

blend weights).

When you use sRGB render targets, the runtime converts the render target color into linear space before it performs blending. The runtime converts the final blended value back into sRGB space before it saves the value back to the render target.

Differences between Direct3D 9 and Direct3D 10: In Direct3D 9, fixed-function blending can be enabled independently for each render target.

In Direct3D 10 and higher, there is one blend-state description; therefore, one blending value can be set for all render targets.

Dual-Source Color Blending

This feature enables the output-merger stage to simultaneously use both pixel shader outputs (o_0 and o_1) as inputs to a blending operation with the single render target at slot 0. Valid blend operations include: add, subtract and revsubtract. Valid blend options for SrcBlend, DestBlend, SrcBlendAlpha or DestBlendAlpha include: `D3D11_BLEND_SRC1_COLOR`, `D3D11_BLEND_INV_SRC1_COLOR`, `D3D11_BLEND_SRC1_ALPHA`, `D3D11_BLEND_INV_SRC1_ALPHA`. The blend equation and the output write mask specify which components the pixel shader is outputting. Extra components are ignored.

Writing to other pixel shader outputs (o_2 , o_3 etc.) is undefined; you may not write to a render target if it is not bound to slot 0. Writing o_{Depth} is valid during dual source color blending.

For examples, see [blending pixel shader outputs](#).

Multiple RenderTargets Overview

A pixel shader can be used to render to at least 8 separate render targets, all of which must be the same type (buffer, Texture1D, Texture1DArray, and so on). Furthermore, all render targets must have the same size in all dimensions (width, height, depth, array size, sample counts). Each render target may have a different data format.

You may use any combination of render targets slots (up to 8). However, a resource view cannot be bound to multiple render-target-slots simultaneously. A view may be reused as long as the resources are not used simultaneously.

Output-Write Mask Overview

Use an output-write mask to control (per component) what data can be written to a render target.

Sample Mask Overview

A sample mask is a 32-bit multisample coverage mask that determines which samples get updated in active render targets. Only one sample mask is allowed. The mapping of bits in a sample mask to the samples in a resource is defined by a user. For n-sample rendering, the first n bits (from the LSB) of the sample mask are used (32 bits if the maximum number of bits).

In this section

TOPIC	DESCRIPTION
Configuring Depth-Stencil Functionality	This section covers the steps for setting up the depth-stencil buffer, and depth-stencil state for the output-merger stage.

Topic	Description
Configuring Blending Functionality	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.
Depth Bias	Polygons that are coplanar in 3D space can be made to appear as if they are not coplanar by adding a z-bias (or depth bias) to each one.

Related topics

[Graphics Pipeline](#)

[Pipeline Stages \(Direct3D 10\)](#)

Configuring Depth-Stencil Functionality

11/2/2020 • 8 minutes to read • [Edit Online](#)

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 Depth-Stencil State](#)
- [Bind Depth-Stencil Data to the OM Stage](#)

Once you know how to use the depth-stencil buffer and the corresponding depth-stencil state, refer to [advanced-stencil techniques](#).

Create a Depth-Stencil Resource

Create the depth-stencil buffer using a texture resource.

```
ID3D11Texture2D* pDepthStencil = NULL;
D3D11_TEXTURE2D_DESC descDepth;
descDepth.Width = backBufferSurfaceDesc.Width;
descDepth.Height = backBufferSurfaceDesc.Height;
descDepth.MipLevels = 1;
descDepth.ArraySize = 1;
descDepth.Format = pDeviceSettings->d3d11.AutoDepthStencilFormat;
descDepth.SampleDesc.Count = 1;
descDepth.SampleDesc.Quality = 0;
descDepth.Usage = D3D11_USAGE_DEFAULT;
descDepth.BindFlags = D3D11_BIND_DEPTH_STENCIL;
descDepth.CPUAccessFlags = 0;
descDepth.MiscFlags = 0;
hr = pd3dDevice->CreateTexture2D( &descDepth, NULL, &pDepthStencil );
```

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.

```

D3D11_DEPTH_STENCIL_DESC dsDesc;

// Depth test parameters
dsDesc.DepthEnable = true;
dsDesc.DepthWriteMask = D3D11_DEPTH_WRITE_MASK_ALL;
dsDesc.DepthFunc = D3D11_COMPARISON_LESS;

// Stencil test parameters
dsDesc.StencilEnable = true;
dsDesc.StencilReadMask = 0xFF;
dsDesc.StencilWriteMask = 0xFF;

// Stencil operations if pixel is front-facing
dsDesc.FrontFace.StencilFailOp = D3D11_STENCIL_OP_KEEP;
dsDesc.FrontFace.StencilDepthFailOp = D3D11_STENCIL_OP_INCR;
dsDesc.FrontFace.StencilPassOp = D3D11_STENCIL_OP_KEEP;
dsDesc.FrontFace.StencilFunc = D3D11_COMPARISON_ALWAYS;

// Stencil operations if pixel is back-facing
dsDesc.BackFace.StencilFailOp = D3D11_STENCIL_OP_KEEP;
dsDesc.BackFace.StencilDepthFailOp = D3D11_STENCIL_OP_DECR;
dsDesc.BackFace.StencilPassOp = D3D11_STENCIL_OP_KEEP;
dsDesc.BackFace.StencilFunc = D3D11_COMPARISON_ALWAYS;

// Create depth stencil state
ID3D11DepthStencilState * pDSState;
pd3dDevice->CreateDepthStencilState(&dsDesc, &pDSState);

```

DepthEnable and StencilEnable enable (and disable) depth and stencil testing. Set DepthEnable to **FALSE** to disable depth testing and prevent writing to the depth buffer. Set StencilEnable to **FALSE** to disable stencil testing and prevent writing to the stencil buffer (when DepthEnable is **FALSE** and StencilEnable is **TRUE**, the depth test always passes in the stencil operation).

DepthEnable only affects the output-merger stage - it does not affect clipping, depth bias, or clamping of values before the data is input to a pixel shader.

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.

```

D3D11_DEPTH_STENCIL_VIEW_DESC descDSV;
descDSV.Format = DXGI_FORMAT_D32_FLOAT_S8X24_UINT;
descDSV.ViewDimension = D3D11_DSV_DIMENSION_TEXTURE2D;
descDSV.Texture2D.MipSlice = 0;

// Create the depth stencil view
ID3D11DepthStencilView* pDSV;
hr = pd3dDevice->CreateDepthStencilView( pDepthStencil, // Depth stencil texture
                                            &descDSV, // Depth stencil desc
                                            &pDSV ); // [out] Depth stencil view

// Bind the depth stencil view
pd3dDeviceContext->OMSetRenderTargets( 1, // One rendertarget view
                                         &pRTV, // Render target view, created earlier
                                         pDSV ); // Depth stencil view for the render target

```

An array of render-target views may be passed into [ID3D11DeviceContext::OMSetRenderTargets](#), however all of those render-target views will correspond to a single depth stencil view. The render target array in Direct3D 11 is a feature that enables an application to render onto multiple render targets simultaneously at the primitive level. Render target arrays offer increased performance over individually setting render targets with multiple calls to [ID3D11DeviceContext::OMSetRenderTargets](#) (essentially the method employed in Direct3D 9).

Render targets must all be the same type of resource. If multisample antialiasing is used, all bound render targets and depth buffers must have the same sample counts.

When a buffer is used as a render target, depth-stencil testing and multiple render targets are not supported.

- As many as 8 render targets can be bound simultaneously.
- All render targets must have the same size in all dimensions (width and height, and depth for 3D or array size for *Array types).
- Each render target may have a different data format.
- Write masks control what data gets written to a render target. The output write masks control on a per-render target, per-component level what data gets written to the render target(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](#)
- [Decaling](#)
- [Outlines and Silhouettes](#)
- [Two-Sided Stencil](#)
- [Reading the Depth-Stencil Buffer as a Texture](#)

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](#).

Reading the Depth-Stencil Buffer as a Texture

An inactive depth-stencil buffer can be read by a shader as a texture. An application that reads a depth-stencil buffer as a texture renders in two passes, the first pass writes to the depth-stencil buffer and the second pass reads from the buffer. This allows a shader to compare depth or stencil values previously written to the buffer against the value for the pixel currently being rendered. The result of the comparison can be used to create effects such as shadow mapping or soft particles in a particle system.

To create a depth-stencil buffer that can be used as both a depth-stencil resource and a shader resource a few changes need to be made to sample code in the [Create a Depth-Stencil Resource](#) section.

- The depth-stencil resource must have a typeless format such as DXGI_FORMAT_R32_TYPELESS.

```
descDepth.Format = DXGI_FORMAT_R32_TYPELESS;
```

- The depth-stencil resource must use both the D3D10_BIND_DEPTH_STENCIL and D3D10_BIND_SHADER_RESOURCE bind flags.

```
descDepth.BindFlags = D3D10_BIND_DEPTH_STENCIL | D3D10_BIND_SHADER_RESOURCE;
```

In addition a shader resource view needs to be created for the depth buffer using a [D3D11_SHADER_RESOURCE_VIEW_DESC](#) structure and [ID3D11Device::CreateShaderResourceView](#). The shader resource view will use a typed format such as DXGI_FORMAT_R32_FLOAT that is the equivalent to

the typeless format specified when the depth-stencil resource was created.

In the first render pass the depth buffer is bound as described in the [Bind Depth-Stencil Data to the OM Stage](#) section. Note that the format passed to [D3D11_DEPTH_STENCIL_VIEW_DESC](#).Format will use a typed format such as DXGI_FORMAT_D32_FLOAT. After the first render pass the depth buffer will contain the depth values for the scene.

In the second render pass the [ID3D11DeviceContext::OMSetRenderTargets](#) function is used to set the depth-stencil view to **NULL** or a different depth-stencil resource and the shader resource view is passed to the shader using [ID3D11EffectShaderResourceVariable::SetResource](#). This allows the shader to look up the depth values calculated in the first rendering pass. Note that a transformation will need to be applied to retrieve depth values if the point of view of the first render pass is different from the second render pass. For example, if a shadow mapping technique is being used the first render pass will be from the perspective of a light source while the second render pass will be from the viewer's perspective.

Related topics

[Output-Merger Stage](#)

[Pipeline Stages \(Direct3D 10\)](#)

Configuring Blending Functionality

11/2/2020 • 4 minutes to read • [Edit Online](#)

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](#)
- [Bind the Blend State](#)
- [Advanced Blending Topics](#)
 - [Alpha-To-Coverage](#)
 - [Blending Pixel Shader Outputs](#)
- [Related topics](#)

Create the Blend State

The blend state is a collection of states used to control blending. These states (defined in [D3D11_BLEND_DESC1](#)) are used to create the blend state object by calling [ID3D11Device1::CreateBlendState1](#).

For instance, here is a very simple example of blend-state creation that disables alpha blending and uses no per-component pixel masking.

```
ID3D11BlendState1* g_pBlendStateNoBlend = NULL;

D3D11_BLEND_DESC1 BlendState;
ZeroMemory(&BlendState, sizeof(D3D11_BLEND_DESC1));
BlendState.RenderTarget[0].BlendEnable = FALSE;
BlendState.RenderTarget[0].RenderTargetWriteMask = D3D11_COLOR_WRITE_ENABLE_ALL;
pd3dDevice->CreateBlendState1(&BlendState, &g_pBlendStateNoBlend);
```

This example is similar to the [HLSLWithoutFX10 Sample](#).

Bind the Blend State

After you create the blend-state object, bind the blend-state object to the output-merger stage by calling [ID3D11DeviceContext::OMSetBlendState](#).

```
float blendFactor[4] = { 0.0f, 0.0f, 0.0f, 0.0f };
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 can pass in **NULL** for the blend-state object to specify the default blend state or pass in a blend-state object. If you created the blend-state object with [D3D11_BLEND_BLEND_FACTOR](#) or [D3D11_BLEND_INV_BLEND_FACTOR](#), you can pass a blend factor to modulate values for the pixel shader, render target, or both. If you didn't create the blend-state object with [D3D11_BLEND_BLEND_FACTOR](#) or [D3D11_BLEND_INV_BLEND_FACTOR](#), you can still pass a non-NULL blend factor, but the blending stage does not use the blend factor; the runtime stores the blend factor, and you can later call

[ID3D11DeviceContext::OMGetBlendState](#) to retrieve the blend factor. If you pass **NULL**, the runtime uses or stores a blend factor equal to { 1, 1, 1, 1 }. The sample mask is a user-defined mask that determines how to sample the existing render target before updating it. The default sampling mask is 0xffffffff which designates point sampling.

In most depth buffering schemes, the pixel closest to the camera is the one that gets drawn. When [setting up the depth stencil state](#), the **DepthFunc** member of [D3D11_DEPTH_STENCIL_DESC](#) can be any [D3D11_COMPARISON_FUNC](#). Normally, you would want **DepthFunc** to be [D3D11_COMPARISON_LESS](#), so that the pixels closest to the camera will overwrite the pixels behind them. However, depending on the needs of your application, any of the other comparison functions may be used to do the depth test.

Advanced Blending Topics

- [Alpha-To-Coverage](#)
- [Blending Pixel Shader Outputs](#)

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 that use alpha transparency to define edges within the surface.

You can use the **AlphaToCoverageEnable** member of [D3D11_BLEND_DESC1](#) or [D3D11_BLEND_DESC](#) to toggle whether the runtime converts the .a component (alpha) of output register [SV_Target0](#) from the pixel shader to an n-step coverage mask (given an n-sample [RenderTarget](#)). The runtime performs an **AND** operation of this mask with the typical sample coverage for the pixel in the primitive (in addition to the sample mask) to determine which samples to update in all the active [RenderTargets](#).

If the pixel shader outputs [SV_Coverage](#), the runtime disables alpha-to-coverage.

NOTE

In multisampling, the runtime shares only one coverage for all [RenderTarget](#)s. The fact that the runtime reads and converts .a from output [SV_Target0](#) to coverage when **AlphaToCoverageEnable** is TRUE does not change the .a value that goes to the blender at [RenderTarget](#) 0 (if a [RenderTarget](#) happens to be set there). In general, if you enable alpha-to-coverage, you don't affect how all color outputs from pixel shaders interact with [RenderTarget](#)s through the [output-merger stage](#) except that the runtime performs an **AND** operation of the coverage mask with the alpha-to-coverage mask. Alpha-to-coverage works independently to whether the runtime can blend [RenderTarget](#) or whether you use blending on [RenderTarget](#).

Graphics hardware doesn't precisely specify exactly how it converts pixel shader [SV_Target0.a](#) (alpha) to a coverage mask, except that alpha of 0 (or less) must map to no coverage and alpha of 1 (or greater) must map to full coverage (before the runtime performs an **AND** operation with actual primitive coverage). As alpha goes from 0 to 1, the resulting coverage should generally increase monotonically. However, hardware might perform area dithering to provide some better quantization of alpha values at the cost of spatial resolution and noise. An alpha value of NaN (Not a Number) results in a no coverage (zero) mask.

Alpha-to-coverage is also traditionally used for screen-door transparency or defining detailed silhouettes for otherwise opaque sprites.

Blending Pixel Shader Outputs

This feature enables the output merger to use both the pixel shader outputs simultaneously as input sources to a blending operation with the single render target at slot 0.

This example takes two results and combines them in a single pass, blending one into the destination with a

multiply and the other with an add:

```
SrcBlend = D3D11_BLEND_ONE;  
DestBlend = D3D11_BLEND_SRC1_COLOR;
```

This example configures the first pixel shader output as the source color and the second output as a per-color component blend factor.

```
SrcBlend = D3D11_BLEND_SRC1_COLOR;  
DestBlend = D3D11_BLEND_INV_SRC1_COLOR;
```

This example illustrates how the blend factors must match the shader swizzles:

```
SrcFactor = D3D11_BLEND_SRC1_ALPHA;  
DestFactor = D3D11_BLEND_SRC_COLOR;  
OutputWriteMask[0] = .ra; // pseudocode for setting the mask at  
// RenderTarget slot 0 to .ra
```

Together, the blend factors and the shader code imply that the pixel shader is required to output at least o0.r and o1.a. Extra output components can be output by the shader but would be ignored, fewer components would produce undefined results.

Related topics

[Output-Merger Stage](#)

[Pipeline Stages \(Direct3D 10\)](#)

Depth Bias

2/4/2021 • 2 minutes to read • [Edit Online](#)

Polygons that are coplanar in 3D space can be made to appear as if they are not coplanar by adding a z-bias (or depth bias) to each one.

This is a technique commonly used to ensure that shadows in a scene are displayed properly. For instance, a shadow on a wall will likely have the same depth value as the wall. If an application renders a wall first and then a shadow, the shadow might not be visible, or depth artifacts might be visible.

An application can help ensure that coplanar polygons are rendered properly by adding the bias (from the **DepthBias** member of [D3D11_RASTERIZER_DESC1](#)) to the z-values that the system uses when rendering the sets of coplanar polygons. Polygons with a larger z value will be drawn in front of polygons with a smaller z value.

There are two options for calculating depth bias.

1. If the depth buffer currently bound to the output-merger stage has a **UNORM** format or no depth buffer is bound, the bias value is calculated like this:

```
Bias = (float)DepthBias * r + SlopeScaledDepthBias * MaxDepthSlope;
```

where r is the minimum representable value > 0 in the depth-buffer format converted to **float32**. The **DepthBias** and **SlopeScaledDepthBias** values are [D3D11_RASTERIZER_DESC1](#) structure members. The **MaxDepthSlope** value is the maximum of the horizontal and vertical slopes of the depth value at the pixel.

2. If a floating-point depth buffer is bound to the output-merger stage the bias value is calculated like this:

```
Bias = (float)DepthBias * 2**exponent(max z in primitive) - r +  
SlopeScaledDepthBias * MaxDepthSlope;
```

where r is the number of mantissa bits in the floating point representation (excluding the hidden bit); for example, 23 for **float32**.

The bias value is then clamped like this:

```
if(DepthBiasClamp > 0)  
    Bias = min(DepthBiasClamp, Bias)  
else if(DepthBiasClamp < 0)  
    Bias = max(DepthBiasClamp, Bias)
```

The bias value is then used to calculate the pixel depth.

```
if ( (DepthBias != 0) || (SlopeScaledDepthBias != 0) )  
    z = z + Bias
```

Depth-bias operations occur on vertices after clipping, therefore, depth-bias has no effect on geometric clipping. The bias value is constant for a given primitive and is added to the z value for each vertex before interpolator setup. When you use [feature levels](#) 10.0 and higher, all bias calculations are performed using 32-bit floating-point arithmetic. Bias is not applied to any point or line primitives, except for lines drawn in wireframe mode.

One of the artifacts with shadow buffer based shadows is shadow acne, or a surface shadowing itself due to minor differences between the depth computation in a shader, and the depth of the same surface in the shadow buffer. One way to alleviate this is to use **DepthBias** and **SlopeScaledDepthBias** when rendering a shadow buffer. The idea is to push surfaces out enough while rendering a shadow buffer so that the comparison result (between the shadow buffer z and the shader z) is consistent across the surface, and avoid local self-shadowing.

However, using **DepthBias** and **SlopeScaledDepthBias** can introduce new rendering problems when a polygon viewed at an extremely sharp angle causes the bias equation to generate a very large z value. This in effect pushes the polygon extremely far away from the original surface in the shadow map. One way to help alleviate this particular problem is to use **DepthBiasClamp**, which provides an upper bound (positive or negative) on the magnitude of the z bias calculated.

NOTE

For [feature levels](#) 9.1, 9.2, 9.3, **DepthBiasClamp** is not supported.

Related topics

[Output-Merger Stage](#)

Compute Shader Overview

11/2/2020 • 2 minutes to read • [Edit Online](#)

A compute shader is a programmable shader stage that expands Microsoft Direct3D 11 beyond graphics programming. The compute shader technology is also known as the DirectCompute technology.

Like other programmable shaders (vertex and geometry shaders for example), a compute shader is designed and implemented with [HLSL](#) but that is just about where the similarity ends. A compute shader provides high-speed general purpose computing and takes advantage of the large numbers of parallel processors on the graphics processing unit (GPU). The compute shader provides memory sharing and thread synchronization features to allow more effective parallel programming methods. You call the [ID3D11DeviceContext::Dispatch](#) or [ID3D11DeviceContext::DispatchIndirect](#) method to execute commands in a compute shader. A compute shader can run on many threads in parallel.

Using Compute Shader on Direct3D 10.x Hardware

A compute shader on Microsoft Direct3D 10 is also known as DirectCompute 4.x.

If you use the Direct3D 11 API and updated drivers, [feature level](#) 10 and 10.1 Direct3D hardware can optionally support a limited form of DirectCompute that uses the cs_4_0 and cs_4_1 [profiles](#). When you use DirectCompute on this hardware, keep the following limitations in mind:

- The maximum number of threads is limited to D3D11_CS_4_X_THREAD_GROUP_MAX_THREADS_PER_GROUP (768) per group.
- The X and Y dimension of **numthreads** is limited to D3D11_CS_4_X_THREAD_GROUP_MAX_X (768) and D3D11_CS_4_X_THREAD_GROUP_MAX_Y (768).
- The Z dimension of **numthreads** is limited to 1.
- The Z dimension of dispatch is limited to D3D11_CS_4_X_DISPATCH_MAX_THREAD_GROUPS_IN_Z_DIMENSION (1).
- Only one unordered-access view can be bound to the shader (D3D11_CS_4_X_UAV_REGISTER_COUNT is 1).
- Only [RWStructuredBuffers](#) and [RWByteAddressBuffers](#) are available as unordered-access views.
- A thread can only access its own region in groupshared memory for writing, though it can read from any location.
- [SV_GroupIndex](#) or [SV_DispatchThreadID](#) must be used when accessing **groupshared** memory for writing.
- **Groupshared** memory is limited to 16KB per group.
- A single thread is limited to a 256 byte region of **groupshared** memory for writing.
- No atomic instructions are available.
- No double-precision values are available.

Using Compute Shader on Direct3D 11.x Hardware

A compute shader on Direct3D 11 is also known as DirectCompute 5.0.

When you use DirectCompute with cs_5_0 [profiles](#), keep the following items in mind:

- The maximum number of threads is limited to D3D11_CS_THREAD_GROUP_MAX_THREADS_PER_GROUP (1024) per group.
- The X and Y dimension of **numthreads** is limited to D3D11_CS_THREAD_GROUP_MAX_X (1024) and D3D11_CS_THREAD_GROUP_MAX_Y (1024).

- The Z dimension of **numthreads** is limited to D3D11_CS_THREAD_GROUP_MAX_Z (64).
- The maximum dimension of dispatch is limited to D3D11_CS_DISPATCH_MAX_THREAD_GROUPS_PER_DIMENSION (65535).
- The maximum number of unordered-access views that can be bound to a shader is D3D11_PS_CS_UAV_REGISTER_COUNT (8).
- Supports [RWStructuredBuffers](#), [RWByteAddressBuffers](#), and typed unordered-access views ([RWTexture1D](#), [RWTexture2D](#), [RWTexture3D](#), and so on).
- Atomic instructions are available.
- Double-precision support might be available. For information about how to determine whether double-precision is available, see [D3D11_FEATURE_DOUBLES](#).

In this section

TOPIC	DESCRIPTION
New Resource Types	Several new resource types have been added in Direct3D 11.
Accessing Resources	There are several ways to access resources .
Atomic Functions	To access a new resource type or shared memory, use an interlocked intrinsic function. Interlocked functions are guaranteed to operate atomically. That is, they are guaranteed to occur in the order programmed. This section lists the atomic functions.

Related topics

[Graphics Pipeline](#)

[How To: Create a Compute Shader](#)

New Resource Types

11/2/2020 • 2 minutes to read • [Edit Online](#)

Several new resource types have been added in Direct3D 11.

- [Read/Write Buffers and Textures](#)
- [Structured Buffer](#)
- [Byte Address Buffer](#)
- [Unordered Access Buffer or Texture](#)
 - [Append and Consume Buffer](#)
- [Related topics](#)

Read/Write Buffers and Textures

Shader model 4 resources are read only. Shader model 5 implements a new corresponding set of read/write resources:

- [RWBuffer](#)
- [RWTexture1D, RWTexture1DArray](#)
- [RWTexture2D, RWTexture2DArray](#)
- [RWTexture3D](#)

These resources require a resource variable to access memory (through indexing) as there are no methods for accessing memory directly. A resource variable can be used on the left and right sides of an equation; if used on the right side, the template type must be single component (float, int, or uint).

Structured Buffer

A structured buffer is a buffer that contains elements of equal sizes. Use a structure with one or more member types to define an element. Here is a structure with three members.

```
struct MyStruct
{
    float4 Color;
    float4 Normal;
    bool isAwesome;
};
```

Use this structure to declare a structured buffer like this:

```
StructuredBuffer<MyStruct> mySB;
```

In addition to indexing, a structured buffer supports accessing a single member like this:

```
float4 myColor = mySB[27].Color;
```

Use the following object types to access a structured buffer:

- [StructuredBuffer](#) is a read only structured buffer.

- [RWStructuredBuffer](#) is a read/write structured buffer.

[Atomic functions](#) which implement interlocked operations are allowed on int and uint elements in an [RWStructuredBuffer](#).

Byte Address Buffer

A byte address buffer is a buffer whose contents are addressable by a byte offset. Normally, the contents of a [buffer](#) are indexed per element using a stride for each element (S) and the element number (N) as given by S*N. A byte address buffer, which can also be called a raw buffer, uses a byte value offset from the beginning of the buffer to access data. The byte value must be a multiple of four so that it is DWORD aligned. If any other value is provided, behavior is undefined.

Shader model 5 introduces objects for accessing a [read-only byte address buffer](#) as well as a [read-write byte address buffer](#). The contents of a byte address buffer is designed to be a 32-bit unsigned integer; if the value in the buffer is not really an unsigned integer, use a function such as [asfloat](#) to read it.

Unordered Access Buffer or Texture

An unordered access resource (which includes buffers, textures, and texture arrays - without multisampling), allows temporally unordered read/write access from multiple threads. This means that this resource type can be read/written simultaneously by multiple threads without generating memory conflicts through the use of [Atomic Functions](#).

Create an unordered access buffer or texture by calling a function such as [ID3D11Device::CreateBuffer](#) or [ID3D11Device::CreateTexture2D](#) and passing in the D3D11_BIND_UNORDERED_ACCESS flag from the [D3D11_BIND_FLAG](#) enumeration.

Unordered access resources can only be bound to pixel shaders and compute shaders. During execution, pixel shaders or compute shaders running in parallel have the same unordered access resources bound.

Append and Consume Buffer

An append and consume buffer is a special type of an unordered resource that supports adding and removing values from the end of a buffer similar to the way a stack works.

An append and consume buffer must be a structured buffer:

- [AppendStructuredBuffer](#)
- [ConsumeStructuredBuffer](#)

Use these resources through their methods, these resources do not use resource variables.

Related topics

[Compute Shader Overview](#)

Accessing Resources

11/2/2020 • 2 minutes to read • [Edit Online](#)

There are several ways to access [resources](#). Regardless, Direct3D guarantees to return zero for any resource that is accessed out of bounds.

- [Access By Byte Offset](#)
- [Access By Index](#)
- [Access By Mips Method](#)
- [Related topics](#)

Access By Byte Offset

Two new buffer types can be accessed using a byte offset:

- [ByteAddressBuffer](#) is a read-only buffer.
- [RWByteAddressBuffer](#) is a read or write buffer.

Access By Index

Resource types can use an index to reference a specific location in the resource. Consider this example:

```
uint2 pos;
Texture2D<float4> myTexture;
float4 myVar = myTexture[pos];
```

This example assigns the 4 float values that are stored at the texel located at the *pos* position in the *myTexture* 2D texture resource to the *myVar* variable.

NOTE

The default for accessing a texture in this way is mipmap level zero (the most detailed level).

NOTE

The "float4 myVar = myTexture[pos];" line is equivalent to "float4 myVar = myTexture.Load(uint3(pos,0));". Access by index is a new HLSL syntax enhancement.

NOTE

The compiler in the June 2010 version of the DirectX SDK and later lets you index all resource types except for [byte address buffers](#).

NOTE

The June 2010 compiler and later lets you declare local resource variables. You can assign globally defined resources (like *myTexture*) to these variables and use them the same way as their global counterparts.

Access By Mips Method

Texture objects have a **mips** method (for example, [Texture2D.mips](#)), which you can use to specify the mipmap level. This example reads the color stored at (7,16) on mipmap level 2 in a 2D texture:

```
uint x = 7;
uint y = 16;
float4 myColor = myTexture.mips[2][uint2(x,y)];
```

This is an enhancement from the June 2010 compiler and later. The "myTexture.mips[2][uint2(x,y)]" expression is equivalent to "myTexture.Load(uint3(x,y,2))".

Related topics

[Compute Shader Overview](#)

Atomic Functions

11/2/2020 • 2 minutes to read • [Edit Online](#)

To access a new resource type or shared memory, use an interlocked intrinsic function. Interlocked functions are guaranteed to operate atomically. That is, they are guaranteed to occur in the order programmed. This section lists the atomic functions.

- [InterlockedAdd](#)
- [InterlockedMin](#)
- [InterlockedMax](#)
- [InterlockedOr](#)
- [InterlockedAnd](#)
- [InterlockedXor](#)
- [InterlockedCompareStore](#)
- [InterlockedCompareExchange](#)
- [InterlockedExchange](#)

Related topics

[Compute Shader Overview](#)

Rendering (Direct3D 11 Graphics)

2/4/2021 • 2 minutes to read • [Edit Online](#)

This section contains information about several Direct3D 11 rendering technologies.

In this section

TOPIC	DESCRIPTION
MultiThreading	Direct3D 11 implements support for object creation and rendering using multiple threads.
Multiple-Pass Rendering	Multiple-pass rendering is a process in which an application traverses its scene graph multiple times in order to produce an output to render to the display. Multiple-pass rendering improves performance because it breaks up complex scenes into tasks that can run concurrently.

Related topics

[Programming Guide for Direct3D 11](#)

MultiThreading

2/22/2020 • 2 minutes to read • [Edit Online](#)

Direct3D 11 implements support for object creation and rendering using multiple threads.

In this section

TOPIC	DESCRIPTION
Introduction to Multithreading in Direct3D 11	Multithreading is designed to improve performance by performing work using one or more threads at the same time.
Object Creation with Multithreading	Use the ID3D11Device interface to create resources and objects, use the ID3D11DeviceContext for rendering .
Immediate and Deferred Rendering	Direct3D 11 supports two types of rendering: immediate and deferred. Both are implemented by using the ID3D11DeviceContext interface.
Command List	A command list is a sequence of GPU commands that can be recorded and played back. A command list may improve performance by reducing the amount of overhead generated by the runtime.
Threading Differences between Direct3D Versions	Many multi-threaded programming models make use of synchronization primitives (such as mutexes) to create critical sections and prevent code from being accessed by more than one thread at a time.

Related topics

[How To: Check for Driver Support](#)

[Rendering](#)

Introduction to Multithreading in Direct3D 11

11/2/2020 • 2 minutes to read • [Edit Online](#)

Multithreading is designed to improve performance by performing work using one or more threads at the same time.

In the past, this has often been done by generating a single main thread for rendering and one or more threads for doing preparation work such as object creation, loading, processing, and so on. However, with the built in synchronization in Direct3D 11, the goal behind multithreading is to utilize every CPU and GPU cycle without making a processor wait for another processor (particularly not making the GPU wait because it directly impacts frame rate). By doing so, you can generate the most amount of work while maintaining the best frame rate. The concept of a single frame for rendering is no longer as necessary since the API implements synchronization.

Multithreading requires some form of synchronization. For example, if multiple threads that run in an application must access a single device context ([ID3D11DeviceContext](#)), that application must use some synchronization mechanism, such as critical sections, to synchronize access to that device context. This is because processing of the render commands (generally done on the GPU) and generating the render commands (generally done on the CPU through object creation, data loading, state changing, data processing) often use the same resources (textures, shaders, pipeline state, and so on). Organizing the work across multiple threads requires synchronization to prevent one thread from modifying or reading data that is being modified by another thread.

While the use of a device context ([ID3D11DeviceContext](#)) is not thread-safe, the use of a Direct3D 11 device ([ID3D11Device](#)) is thread-safe. Because each [ID3D11DeviceContext](#) is single threaded, only one thread can call a [ID3D11DeviceContext](#) at a time. If multiple threads must access a single [ID3D11DeviceContext](#), they must use some synchronization mechanism, such as critical sections, to synchronize access to that [ID3D11DeviceContext](#). However, multiple threads are not required to use critical sections or synchronization primitives to access a single [ID3D11Device](#). Therefore, if an application uses [ID3D11Device](#) to create resource objects, that application is not required to use synchronization to create multiple resource objects at the same time.

Multithreading support divides the API into two distinct functional areas:

- [Object Creation with Multithreading](#)
- [Immediate and Deferred Rendering](#)

Multithreading performance depends on the driver support. [How To: Check for Driver Support](#) provides more information about querying the driver and what the results mean.

Direct3D 11 has been designed from the ground up to support multithreading. Direct3D 10 implements limited support for multithreading using the [thread-safe layer](#). This page lists the behavior differences between the two versions of DirectX: [Threading Differences between Direct3D Versions](#).

Multithreading and DXGI

Only one thread at a time should use the immediate context. However, your application should also use that same thread for Microsoft DirectX Graphics Infrastructure (DXGI) operations, especially when the application makes calls to the [IDXGISwapChain::Present](#) method.

NOTE

It is invalid to use an immediate context concurrently with most of the DXGI interface functions. For the March 2009 and later DirectX SDKs, the only DXGI functions that are safe are [AddRef](#), [Release](#), and [QueryInterface](#).

For more info about using DXGI with multiple threads, see [Multithread Considerations](#).

Related topics

[Multithreading](#)

Object Creation with Multithreading

2/22/2020 • 2 minutes to read • [Edit Online](#)

Use the [ID3D11Device](#) interface to create resources and objects, use the [ID3D11DeviceContext](#) for rendering.

Here are some examples of how to create resources:

- [How to: Create a Vertex Buffer](#)
- [How to: Create an Index Buffer](#)
- [How to: Create a Constant Buffer](#)
- [How to: Initialize a Texture From a File](#)

Multithreading Considerations

The amount of concurrency of resource creation and rendering that your application can achieve depends on the level of multithreading support that the driver implements. If the driver supports concurrent creates, then the application should have much less concurrency concerns. However, if the driver does not support concurrent object creation, the amount of concurrency is extremely limited. To understand the amount of support available in a driver, query the driver for the value of the [DriverConcurrentCreates](#) member of the [D3D11_FEATURE_DATA_THREADING](#) structure. For more information about checking for driver support of multithreading, see [How To: Check for Driver Support](#).

Concurrent operations do not necessarily lead to better performance. For example, creating and loading a texture is typically limited by memory bandwidth. Attempting to create and load multiple textures might be no faster than doing one texture at a time, even if this leaves multiple CPU cores idle. However, creating multiple shaders using multiple threads can increase performance as this operation is less dependent on system resources such as memory bandwidth.

When the driver and graphics hardware support multithreading, you can typically initialize newly created resources more efficiently by passing a pointer to initial data in the *pInitialData* parameter (for example, in a call to the [ID3D11Device::CreateTexture2D](#) method).

Related topics

[MultiThreading](#)

Immediate and Deferred Rendering

2/22/2020 • 2 minutes to read • [Edit Online](#)

Direct3D 11 supports two types of rendering: immediate and deferred. Both are implemented by using the [ID3D11DeviceContext](#) interface.

Immediate Rendering

Immediate rendering refers to calling rendering APIs or commands from a device, which queues the commands in a buffer for execution on the GPU. Use an immediate context to render, set pipeline state, and play back a command list.

Create an immediate context using [D3D11CreateDevice](#) or [D3D11CreateDeviceAndSwapChain](#).

Deferred Rendering

Deferred rendering records graphics commands in a command buffer so that they can be played back at some other time. Use a deferred context to record commands (rendering as well as state setting) to a command list. Deferred rendering is a new concept in Direct3D 11; deferred rendering is designed to support rendering on one thread while recording commands for rendering on additional threads. This parallel, multithread strategy allows you to break up complex scenes into concurrent tasks. For more information about rendering complex scenes, see [Multiple-Pass Rendering](#).

Create a deferred context using [ID3D11Device::CreateDeferredContext](#).

Direct3D generates rendering overhead when it queues up commands in the command buffer. In contrast, a [command list](#) executes much more efficiently during playback. To use a command list, record rendering commands with a deferred context and play them back using an immediate context.

You can generate only a single command list in a single-threaded fashion. However, you can create and use multiple deferred contexts simultaneously, each in a separate thread. Then, you can use those multiple deferred contexts to simultaneously create multiple command lists.

You cannot play back two or more command lists simultaneously on the immediate context.

To determine if a device context is an immediate or a deferred context, call [ID3D11DeviceContext::GetType](#).

The [ID3D11DeviceContext::Map](#) method is only supported in a deferred context for dynamic resources ([D3D11_USAGE_DYNAMIC](#)) where the first call within the command list is [D3D11_MAP_WRITE_DISCARD](#). [D3D11_MAP_WRITE_NO_OVERWRITE](#) is supported on subsequent calls if available for the given type of resource.

Queries in a deferred context are limited to data generation and predicated drawing. You cannot call [ID3D11DeviceContext::GetData](#) on a deferred context to get data about a query; you can only call [GetData](#) on the immediate context to get data about a query. You can set a rendering predicate (a type of query) by calling [ID3D11DeviceContext::SetPredication](#) to use query results on the GPU. You can generate query data through calls to [ID3D11DeviceContext::Begin](#) and [ID3D11DeviceContext::End](#). However, the query data will not be available until you call [ID3D11DeviceContext::ExecuteCommandList](#) on the immediate context to submit the deferred context command list. The query data is then processed by the GPU.

Related topics

Multithreading

Command List

2/22/2020 • 2 minutes to read • [Edit Online](#)

A command list is a sequence of GPU commands that can be recorded and played back. A command list may improve performance by reducing the amount of overhead generated by the runtime.

Use a command list in the following scenarios:

- Within a single frame, render part of the scene on one thread while recording another part of the scene on a second thread. At the end of the frame, play back the recorded command list on the first thread. Use this approach to scale complex rendering tasks across multiple threads or cores.
- Pre-record a command list before you need to render it (for example, while a level is loading) and efficiently play it back later in your scene. This optimization works well when you need to render something often.

A command list is immutable and is designed to be recorded and played back during a single execution of an application. A command list is not designed to be pre-recorded ahead of game execution and loaded from your media as there is no way to persist the list.

A command list must be recorded by a deferred context, but it can only be played back on an immediate context. Deferred contexts can generate command lists concurrently.

- To record a command list, see [How to: Record a Command List](#).
- To play back a command list, see [How to: Play Back a Command List](#).
- When using a command list, performance depends on the amount of support implemented in a driver. To check for driver support, see [How To: Check for Driver Support](#).

Related topics

[Immediate and Deferred Rendering](#)

[MultiThreading](#)

Threading Differences between Direct3D Versions

11/2/2020 • 2 minutes to read • [Edit Online](#)

Many multi-threaded programming models make use of synchronization primitives (such as mutexes) to create critical sections and prevent code from being accessed by more than one thread at a time.

However, the resource creation methods of the [ID3D11Device](#) interface were designed to be re-entrant, without requiring synchronization primitives and critical sections. As a result, the resource creation methods are efficient and easy to use.

Differences between Direct3D 11, 10 and 9:

Direct3D 11 defaults to mostly thread-safe and continues to allow applications to opt-out using `D3D11_CREATE_DEVICE_SINGLETHREADED`. If applications opt-out of being thread-safe, they must adhere to threading rules. The runtime synchronizes threads on behalf of the application allowing concurrent threads to run. In fact, the synchronization in Direct3D 11 is more efficient than using the thread-safe layer in Direct3D 10.

Direct3D 10 can support the execution of only one thread at a time. Direct3D 10 is fully thread safe and allows an application to opt-out of that behavior by using `D3D10_CREATE_DEVICE_SINGLE_THREADED`.

Direct3D 9 does not default to thread safe. However, when you call [CreateDevice](#) or [CreateDeviceEx](#) to create a device, you can specify the `D3DCREATE_MULTITHREADED` flag to make the Direct3D 9 API thread safe. This causes significant synchronization overhead. Therefore, making the Direct3D 9 API thread safe is not recommended because performance can be degraded.

Related topics

[MultiThreading](#)

Multiple-Pass Rendering

2/22/2020 • 2 minutes to read • [Edit Online](#)

Multiple-pass rendering is a process in which an application traverses its scene graph multiple times in order to produce an output to render to the display. Multiple-pass rendering improves performance because it breaks up complex scenes into tasks that can run concurrently.

To perform multiple-pass rendering, you create a deferred context and command list for each additional pass. While the application traverses the scene graph, it records commands (for example, rendering commands such as [Draw](#)) into a deferred context. After the application finishes the traversal, it calls the [FinishCommandList](#) method on the deferred context. Finally, the application calls the [ExecuteCommandList](#) method on the immediate context to execute the commands in each command list.

The following pseudocode shows how to perform multiple-pass rendering:

```
{  
    ImmCtx->SetRenderTarget( pRTViewOfResourceX );  
    DefCtx1->SetTexture( pSRView1OfResourceX );  
    DefCtx2->SetTexture( pSRView2OfResourceX );  
  
    for () // Traverse the scene graph.  
    {  
        ImmCtx->Draw(); // Pass 0: immediate context renders primitives into resource X.  
  
        // The following texturing by the deferred contexts occurs after the  
        // immediate context makes calls to ExecuteCommandList.  
        // Resource X is then completely updated by the immediate context.  
        DefCtx1->Draw(); // Pass 1: deferred context 1 performs texturing from resource X.  
        DefCtx2->Draw(); // Pass 2: deferred context 2 performs texturing from resource X.  
    }  
  
    // Create command lists and record commands into them.  
    DefCtx1->FinishCommandList( &pCL1 );  
    DefCtx2->FinishCommandList( &pCL2 );  
  
    ImmCtx->ExecuteCommandList( pCL1 ); // Execute pass 1.  
    ImmCtx->ExecuteCommandList( pCL2 ); // Execute pass 2.  
}
```

NOTE

The immediate context modifies a resource, which is bound to the immediate context as a render target view (RTV); in contrast, each deferred context simply uses the resource, which is bound to the deferred context as a shader resource view (SRV). For more information about immediate and deferred contexts, see [Immediate and Deferred Rendering](#).

Related topics

[Rendering](#)

Effects (Direct3D 11)

11/2/2020 • 2 minutes to read • [Edit Online](#)

A DirectX effect is a collection of pipeline state, set by expressions written in [HLSL](#) and some syntax that is specific to the effect framework.

After compiling an effect, use the effect framework APIs to render. Effect functionality can range from something as simple as a vertex shader that transforms geometry and a pixel shader that outputs a solid color, to a rendering technique that requires multiple passes, uses every stage of the graphics pipeline, and manipulates shader state as well as the pipeline state not associated with the programmable shaders.

The first step is to organize the state you want to control in an effect. This includes shader state (vertex, hull, domain, geometry, pixel and compute shaders), texture and sampler state used by the shaders, and other non-programmable pipeline state. You can create an effect in memory as a text string, but typically, the size gets large enough that it is handy to store effect state in an effect file (a text file that ends in a .fx extension). To use an effect, you must compile it (to check HLSL syntax as well as effect framework syntax), initialize effect state through API calls, and modify your render loop to call the rendering APIs.

An effect encapsulates all of the render state required by a particular effect into a single rendering function called a technique. A pass is a sub-set of a technique, that contains render state. To implement a multiple pass rendering effect, implement one or more passes within a technique. For example, say you wanted to render some geometry with one set of depth/stencil buffers, and then draw some sprites on top of that. You could implement the geometry rendering in the first pass, and the sprite drawing in the second pass. To render the effect, you simply render both passes in your render loop. You can implement any number of techniques in an effect. Of course, the greater the number of techniques, the greater the compile time for the effect. One way to exploit this functionality is to create effects with techniques that are designed to run on different hardware. This allows an application to gracefully downgrade performance based on the hardware capabilities detected.

A set of techniques can be grouped in a group (which uses the syntax "fxgroup"). Techniques can be grouped in any way. For example, multiple groups could be created, one per material; each material could have a technique for each hardware level; each technique would have a set of passes which define the material on the particular hardware.

In this section

TOPIC	DESCRIPTION
Organizing State in an Effect	With Direct3D 11, effect state for certain pipeline stages is organized by structures.
Effect System Interfaces	The effect system defines several interfaces for managing effect state.
Specializing Interfaces	ID3DX11EffectVariable has a number of methods for casting the interface into the particular type of interface you need.
Interfaces and Classes in Effects	There are many ways to use classes and interfaces in Effects 11.

TOPIC	DESCRIPTION
Rendering an Effect	An effect can be used to store information, or to render using a group of state.
Cloning an Effect	Cloning an effect creates a second, almost identical copy of the effect.
Stream Out Syntax	A geometry shader with stream out is declared with a particular syntax.
Differences Between Effects 10 and Effects 11	This topic shows the differences between Effects 10 and Effects 11.

Related topics

[Programming Guide for Direct3D 11](#)

Organizing State in an Effect (Direct3D 11)

11/2/2020 • 4 minutes to read • [Edit Online](#)

With Direct3D 11, effect state for certain pipeline stages is organized by structures. Here are the structures:

PIPELINE STATE	STRUCTURE
Rasterization	D3D11_RASTERIZER_DESC
Output Merger	D3D11_BLEND_DESC and D3D11_DEPTH_STENCIL_DESC
Shaders	See below

For the shader stages, where the number of state changes need to be more controlled by an application, the state has been divided up into constant buffer state, sampler state, shader resource state, and unordered access view state (for pixel and compute shaders). This allows an application that is carefully designed to update only the state that is changing, which improves performance by reducing the amount of data that needs to be passed to the GPU.

So how do you organize the pipeline state in an effect?

The answer is, the order doesn't matter. Global variables do not have to be located at the top. However, all the samples in the SDK follow the same order, as it is good practice to organize the data the same way. So this is a brief description of the data ordering in the DirectX SDK samples.

- [Global Variables](#)
- [Shaders](#)
- [Groups, Techniques, and Passes](#)

Global Variables

Just like standard C practice, global variables are declared first, at the top of the file. Most often, these are variables that will be initialized by an application, and then used in an effect. Sometimes they are initialized and never changed, other times they are updated every frame. Just like C function scope rules, effect variables declared outside of the scope of effect functions are visible throughout the effect; any variable declared inside of an effect function is only visible within that function.

Here is an example of the variables declared in BasicHLSL10.fx.

```

// Global variables
float4 g_MaterialAmbientColor;           // Material's ambient color

Texture2D g_MeshTexture;                  // Color texture for mesh

float    g_fTime;                         // App's time in seconds
float4x4 g_mWorld;                       // World matrix for object
float4x4 g_mWorldViewProjection;          // World * View * Projection matrix

// Texture samplers
SamplerState MeshTextureSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = Wrap;
    AddressV = Wrap;
};

```

The syntax for effect variables is more fully detailed in [Effect Variable Syntax \(Direct3D 11\)](#). The syntax for effect texture samplers is more fully detailed in [Sampler Type \(DirectX HLSL\)](#).

Shaders

Shaders are small executable programs. You can think of shaders as encapsulating shader state, since the HLSL code implements the shader functionality. The graphics pipeline up to five different kinds of shaders.

- Vertex shaders - Operate on vertex data. One vertex in yields one vertex out.
- Hull shaders - Operate on patch data. Control Point Phase: one invocation yields one control point; For each Fork and Join Phases: one patch yields some amount of patch constant data.
- Domain shaders - Operate on primitive data. One primitive may yield 0, 1, or many primitives out.
- Geometry shaders - Operate on primitive data. One primitive in may yield 0, 1, or many primitives out.
- Pixel shaders - Operate on pixel data. One pixel in yields 1 pixel out (unless the pixel is culled out of a render).

The compute shader pipeline uses one shader:

- Compute shaders - Operate on any kind of data. The output is independent of the number of threads.

Shaders are local functions and follow C style function rules. When an effect is compiled, each shader is compiled and a pointer to each shader function is stored internally. An ID3D11Effect interface is returned when compilation is successful. At this point the compiled effect is in an intermediate format.

To find out more information about the compiled shaders, you will need to use shader reflection. This is essentially like asking the runtime to decompile the shaders, and return information back to you about the shader code.

```

struct VS_OUTPUT
{
    float4 Position : SV_POSITION; // vertex position
    float4 Diffuse : COLOR0; // vertex diffuse color
    float2 TextureUV : TEXCOORD0; // vertex texture coords
};

VS_OUTPUT RenderSceneVS( float4 vPos : POSITION,
                        float3 vNormal : NORMAL,
                        float2 vTexCoord0 : TEXCOORD,
                        uniform int nNumLights,
                        uniform bool bTexture,
                        uniform bool bAnimate )
{
    VS_OUTPUT Output;
    float3 vNormalWorldSpace;

    ....

    return Output;
}

struct PS_OUTPUT
{
    float4 RGBColor : SV_Target; // Pixel color
};

PS_OUTPUT RenderScenePS( VS_OUTPUT In,
                        uniform bool bTexture )
{
    PS_OUTPUT Output;

    if( bTexture )
        Output.RGBColor = g_MeshTexture.Sample(MeshTextureSampler, In.TextureUV) * In.Diffuse;
    ....

    return Output;
}

```

The syntax for effect shaders is more fully detailed in [Effect Function Syntax \(Direct3D 11\)](#).

Groups, Techniques, and Passes

A group is a collection of techniques. A technique is a collection of rendering passes (there must be at least one pass). Each effect pass (which is similar in scope to a single pass in a render loop) defines the shader state and any other pipeline state necessary to render geometry.

Groups are optional. There is a single, unnamed group which encompasses all global techniques. All other groups must be named.

Here is an example of one technique (which includes one pass) from BasicHLSL10.fx.

```
technique10 RenderSceneWithTexture1Light
{
    pass P0
    {
        SetVertexShader( CompileShader( vs_4_0, RenderSceneVS( 1, true, true ) ) );
        SetGeometryShader( NULL );
        SetPixelShader( CompileShader( ps_4_0, RenderScenePS( true ) ) );
    }
}

fxgroup g0
{
    technique11 RunComputeShader
    {
        pass P0
        {
            SetComputeShader( CompileShader( cs_5_0, CS() ) );
        }
    }
}
```

The syntax for effect shaders is more fully detailed in [Effect Technique Syntax \(Direct3D 11\)](#).

Related topics

[Effects \(Direct3D 11\)](#)

Effect System Interfaces (Direct3D 11)

2/22/2020 • 2 minutes to read • [Edit Online](#)

The effect system defines several interfaces for managing effect state. There are two types of interfaces: those used by the runtime to render an effect and reflection interfaces for getting and setting effect variables.

- [Effect Runtime Interfaces](#)
- [Effect Reflection Interfaces](#)

Effect Runtime Interfaces

Use runtime interfaces to render an effect.

RUNTIME INTERFACES	DESCRIPTION
ID3DX11Effect	Collection of one or more groups and techniques for rendering.
ID3DX11EffectPass	A collection of state assignments.
ID3DX11EffectTechnique	A collection of one or more passes.
ID3DX11EffectGroup	A collection of one or more techniques.

Effect Reflection Interfaces

Reflection is implemented in the effect system to support reading (and writing) effect state. There are multiple ways to access effect variables.

Setting Groups of Effect State

Use these interfaces to get and set a group of state.

REFLECTION INTERFACES	DESCRIPTION
ID3DX11EffectBlendVariable	Get and set blend state.
ID3DX11EffectDepthStencilVariable	Get and set depth-stencil state.
ID3DX11EffectRasterizerVariable	Get and set rasterizer state.
ID3DX11EffectSamplerVariable	Get and set sampler state.

Setting Effect Resources

Use these interfaces to get and set resources.

REFLECTION INTERFACES	DESCRIPTION
ID3DX11EffectConstantBuffer	Access data in a texture buffer or constant buffer.
ID3DX11EffectDepthStencilViewVariable	Access data in a depth-stencil resource.
ID3DX11EffectRenderTargetViewVariable	Access data in a render target.
ID3DX11EffectShaderResourceVariable	Access data in a shader resource.
ID3DX11EffectUnorderedAccessViewVariable	Access data in an unordered access view.

Setting Other Effect Variables

Use these interfaces to get and set state by the variable type.

REFLECTION INTERFACES	DESCRIPTION
ID3DX11EffectClassInstanceVariable	Get a class instance.
ID3DX11EffectInterfaceVariable	Get and set an interface.
ID3DX11EffectMatrixVariable	Get and set a matrix.
ID3DX11EffectScalarVariable	Get and set a scalar.
ID3DX11EffectShaderVariable	Get a shader variable.
ID3DX11EffectStringVariable	Get and set a string.
ID3DX11EffectType	Get a variable type.
ID3DX11EffectVectorVariable	Get and set a vector.

All reflection interfaces derive from [ID3DX11EffectVariable](#).

Related topics

[Effects \(Direct3D 11\)](#)

[Programming Guide for Direct3D 11](#)

Specializing Interfaces (Direct3D 11)

2/22/2020 • 2 minutes to read • [Edit Online](#)

ID3DX11EffectVariable has a number of methods for casting the interface into the particular type of interface you need. The methods are of the form *AsType* and include a method for each type of effect variable (such as AsBlend, AsConstantBuffer etc..)

For example, suppose you have an effect with two global variables: time and a world transform.

```
float     g_fTime;
float4x4  g_mWorld;
```

Here is an example that gets these variables:

```
ID3DX11EffectVariable* g_pVariable;
ID3DX11EffectMatrixVariable* g_pmWorld;
ID3DX11EffectScalarVariable* g_pfTime;

g_pVariable = g_pEffect11->GetVariableByName("g_mWorld");
g_pmWorld = g_pVariable->AsMatrix();
g_pVariable = g_pEffect11->GetVariableByName("g_fTime");
g_pfTime = g_pVariable->AsScalar();
```

By specializing the interfaces, you could reduce the code to a single call.

```
g_pmWorld = (g_pEffect11->GetVariableByName("g_mWorld"))->AsMatrix();
g_pfTime = (g_pEffect11->GetVariableByName("g_fTime"))->AsScalar();
```

Interfaces that inherit from **ID3DX11EffectVariable** also have these methods, but they have been designed to return invalid objects; only calls from **ID3DX11EffectVariable** return valid objects. Applications can test the returned object to see if it is valid by calling **ID3DX11EffectVariable::IsValid**.

Related topics

[Effects \(Direct3D 11\)](#)

Interfaces and Classes in Effects

11/2/2020 • 2 minutes to read • [Edit Online](#)

There are many ways to use classes and interfaces in Effects 11. For interface and class syntax, see [Interfaces and Classes](#).

The following sections detail how to specify class instances to a shader which uses interfaces. We will use the following interface and classes in the examples:

```
interface IColor
{
    float4 GetColor();
};

class CRed : IColor
{
    float4 GetColor() { return float4(1,0,0,1); }
};

class CGreen : IColor
{
    float4 GetColor() { return float4(0,1,0,1); }
};

CRed pRed;
CGreen pGreen;
IColor pIColor;
IColor pIColor2 = pRed;
```

Note that interface instances can be initialized to class instances. Arrays of class and interface instances are also supported and they can be initialized as in the following example:

```
CRed pRedArray[2];
IColor pIColor3 = pRedArray[1];
IColor pIColorArray[2] = {pRed, pGreen};
IColor pIColorArray2[2] = pRedArray;
```

Uniform Interface Parameters

Just like other uniform data types, uniform interface parameters must be specified in the `CompileShader` call. Interface parameters can be assigned to global interface instances or global class instances. When assigned to a global interface instance, the shader will have a dependency on the interface instance, which means that it must be set to a class instance. When assigned to global class instances, the compiler specializes the shader (as with other uniform data types) to use that class. This is important for two scenarios:

1. Shaders with a `4_x` target can use interface parameters if these parameters are uniform and assigned to global class instances (so no dynamic linkage is used).
2. Users can decide to have many compiled, specialized shaders with no dynamic linkage or few compiled shaders with dynamic linkage.

```

float4 PSUniform( uniform IColor color ) : SV_Target
{
    return color;
}

technique11
{
    pass
    {
        SetPixelShader( CompileShader( ps_4_0, PSUniform(pRed) ) );
    }
    pass
    {
        SetPixelShader( CompileShader( ps_5_0, PSUniform(pIColor2) ) );
    }
}

```

If pIColor2 remains unchanged through the API, then the previous two passes are functionally equivalent, but the first uses a ps_4_0 static shader while the second uses a ps_5_0 shader with dynamic linkage. If pIColor2 is changed through the effects API (see Setting Class Instances below), then the behavior of the pixel shader in the second pass may change.

Non-Uniform Interface Parameters

Non-uniform interface parameters create interface dependencies for the shaders. When applying a shader with interface parameters, these parameters must be assigned in with the BindInterfaces call. Global interface instances and global class instances can be specified in the BindInterfaces call.

```

float4 PSAbstract( IColor color ) : SV_Target
{
    return color;
}

PixelShader pPSAbstract = CompileShader( ps_5_0, PSAbstract(pRed) );

technique11
{
    pass
    {
        SetPixelShader( BindInterfaces( pPSAbstract, pRed ) );
    }
    pass
    {
        SetPixelShader( BindInterfaces( pPSAbstract, pIColor2 ) );
    }
}

```

If pIColor2 remains unchanged through the API, then the previous two passes are functionally equivalent and both use dynamic linkage. If pIColor2 is changed through the effects API (see Setting Class Instances below), then the behavior of the pixel shader in the second pass may change.

Setting Class Instances

When setting a shader with dynamic shader linkage to the Direct3D 11 device, class instances must also be specified. It is an error to set such a shader with a **NULL** class instance. Therefore, all interface instances which a shader references must have an associated class instance.

The following example shows how to get a class instance variable from an effect and set it to an interface variable:

```
ID3DX11EffectPass* pPass = pEffect->GetTechniqueByIndex(0)->GetPassByIndex(1);

ID3DX11EffectInterfaceVariable* pIface = pEffect->GetVariableByName( "pIColor2" )->AsInterface();
ID3DX11EffectClassInstanceVariable* pCI = pEffect->GetVariableByName( "pGreen" )->AsClassInstance();
pIface->SetClassInstance( pCI );
pPass->Apply( 0, pDeviceContext );

// Apply the same pass with a different class instance
pCI = pEffect->GetVariableByName( "pRedArray" )->GetElement(1)->AsClassInstance();
pIface->SetClassInstance( pCI );
pPass->Apply( 0, pDeviceContext );
```

Related topics

[Effects \(Direct3D 11\)](#)

Rendering an Effect (Direct3D 11)

11/2/2020 • 2 minutes to read • [Edit Online](#)

An effect can be used to store information, or to render using a group of state. Each technique specifies some set of vertex shaders, hull shaders, domain shaders, geometry shaders, pixel shaders, compute shaders, shader state, sampler and texture state, unordered access view state and other pipeline state. So once you have organized state into an effect, you can encapsulate the rendering effect that results from that state by creating and rendering the effect.

There are a few steps for preparing an effect for rendering. The first is compiling, which checks the HLSL like code against the HLSL language syntax and the effect framework rules. You can compile an effect from your application using API calls or you can compile an effect offline using the effect compiler utility `fxc.exe`. Once your effect has successfully compiled, create it by calling a different (but very similar) set of APIs.

After the effect is created, there are two remaining steps for using it. First, you must initialize the effect state values (the values of the effect variables) using a number of methods for setting state if they are not initialized in HLSL. For some variables this can be done once when an effect is created; others must be updated each time your application calls the render loop. Once the effect variables are set, you tell the runtime to render the effect by applying a technique. These topics are all discussed in further detail below.

Naturally there are performance considerations for using effects. These considerations are largely the same if you are not using an effect. Things like minimizing the amount of state changes and organizing the variables by frequency of update. These tactics are used to minimize the amount of data that needs to be sent from the CPU to the GPU, and therefore minimize potential synchronization problems.

In this section

TOPIC	DESCRIPTION
Compile an Effect	After an effect has been authored, the next step is to compile the code to check for syntax problems.
Create an Effect	An effect is created by loading the compiled effect bytecode into the effects framework. Unlike Effects 10, the effect must be compiled before creating the effect. Effects loaded into memory can be created by calling D3DX11CreateEffectFromMemory .
Set Effect State	Some effect constants only need to be initialized. Once initialized, the effect state is set to the device for the entire render loop. Other variables need to be updated each time the render loop is called. The basic code for setting effect variables is shown below, for each of the types of variables.
Apply a Technique	With the constants, textures, and shader state declared and initialized, the only thing left to do is to set the effect state in the device.

Related topics

[Effects \(Direct3D 11\)](#)

Compile an Effect (Direct3D 11)

11/2/2020 • 4 minutes to read • [Edit Online](#)

After an effect has been authored, the next step is to compile the code to check for syntax problems.

You do so by calling one of the compile APIs ([D3DX11CompileFromFile](#), [D3DX11CompileFromMemory](#), or [D3DX11CompileFromResource](#)). These APIs invoke the effect compiler fxc.exe, which compiles HLSL code.

This is why the syntax for code in an effect looks very much like HLSL code. (There are a few exceptions that will be handled later). The effect compiler/hlsl compiler, fxc.exe, is available in the SDK in the utilities folder so that shaders (or effects) can be compiled offline if you choose. See the documentation for running the compiler from the command line.

- [Example](#)
- [Includes](#)
- [Searching for Include Files](#)
- [Macros](#)
- [HLSL Shader Flags](#)
- [FX Flags](#)
- [Checking Errors](#)
- [Related topics](#)

Example

Here's an example of compiling an effect file.

```
WCHAR str[MAX_PATH];
DXUTFindDXSDKMediaFileCch( str, MAX_PATH, L"BasicHLSL10.fx" );

hr = D3DX11CompileFromFile( str, NULL, NULL, pFunctionName, pProfile, D3D10_SHADER_ENABLE_STRICTNESS, NULL,
NULL, &pBlob, &pErrorBlob, NULL );
```

Includes

One parameter of the compile APIs is an include interface. Generate one of these if you want to include a customized behavior when the compiler reads an include file. The compiler executes this custom behavior each time it creates or compiles an effect (that uses the include pointer). To implement customized include behavior, derive a class from the [ID3DInclude](#) interface. This provides your class with two methods: [Open](#) and [Close](#). Implement the custom behavior in these methods.

Searching for Include Files

The pointer that the compiler passes in the *pParentData* parameter to your include handler's [Open](#) method might not point to the container that includes the #include file that the compiler needs to compile your shader code. That is, the compiler might pass **NULL** in *pParentData*. Therefore, we recommend that your include handler search its own list of include locations for content. Your include handler can dynamically add new include locations as it receives those locations in calls to its [Open](#) method.

In the following example, suppose that the shader code's include files are both stored in the *somewhereelse* directory. When the compiler calls the include handler's [Open](#) method to open and read the contents of *somewhereelse\foo.h*, the include handler can save the location of the *somewhereelse* directory. Later, when

the compiler calls the include handler's **Open** method to open and read the contents of *bar.h*, the include handler can automatically search in the *somewhereelse* directory for *bar.h*.

```
Main.hlsl:  
#include "somewhereelse\foo.h"  
  
Foo.h:  
#include "bar.h"
```

Macros

Effect compilation can also take a pointer to macros that are defined elsewhere. For example, suppose you want to modify the effect in BasicHLSL10, to use two macros: zero and one. The effect code that uses the two macros is shown here.

```
if( bAnimate )  
    vAnimatedPos += float4(vNormal, zero) *  
        (sin(g_fTime+5.5)+0.5)*5;  
  
Output.Diffuse.a = one;
```

Here is the declaration for the two macros.

```
D3D10_SHADER_MACRO Shader_Macros[3] = { "zero", "0", "one", "1.0f", NULL, NULL };
```

The macros are a NULL-terminated array of macros; where each macro is defined by using a [D3D10_SHADER_MACRO](#) struct.

Modify the compile effect call to take a pointer to the macros.

```
D3DX11CompileFromFile( str, Shader_Macros, NULL, pFunctionName,  
                      pProfile, D3D10_SHADER_ENABLE_STRICTNESS, NULL,  
                      NULL, &pBlob, &pErrorBlob, NULL );
```

HLSL Shader Flags

Shader flags specify shader constraints to the HLSL compiler. These flags affect the code generated by the shader compiler in the following ways:

- Optimize the code size.
- Including debug information, which prevents flow control.
- Affects the compile target and whether a shader can run on legacy hardware.

These flags can be logically combined if you have not specified two conflicting characteristics. For a listing of the flags see [D3D10_SHADER Constants](#).

FX Flags

Use these flags when you create an effect to define either compilation behavior or runtime effect behavior. For a listing of the flags see [D3D10_EFFECT Constants](#).

Checking Errors

If during compilation an error occurs, the API returns an interface that contains the errors from the effect compiler. This interface is called **ID3DBlob**. It is not directly readable; however, by returning a pointer to the buffer that contains the data (which is a string), you can see any compilation errors.

This example contains an error in the BasicHLSL.fx, the first variable declaration occurs twice.

```
//-----
// Global variables
//-----
float4 g_MaterialAmbientColor;      // Material's ambient color

// Declare the same variable twice
float4 g_MaterialAmbientColor;      // Material's ambient color
```

This error causes the compiler to return the following error, as shown in the following screen shot of the Watch window in Microsoft Visual Studio.

Watch 1	
Name	Value
(char*)l_pError	0x01997fb8 "BasicHLSL10.fx(18): error X3003: redefinition of 'g_MaterialAmbientColor'"

Because the compiler returns the error in a LPVOID pointer, cast it to a character string in the Watch window.

Here is the code that returns the error from the failed compile.

```
// Read the D3DX effect file
WCHAR str[MAX_PATH];
ID3DBlob* l_pBlob_Effect = NULL;
ID3DBlob* l_pBlob_Errors = NULL;
hr = DXUTFindDXSDKMediaFileCch( str, MAX_PATH, L"BasicHLSL10.fx" );
hr = D3DX11CompileFromFile( str, NULL, NULL, pFunctionName,
                           pProfile, D3D10_SHADER_ENABLE_STRICTNESS, NULL,
                           NULL, &pBlob, &pErrorBlob, NULL );

LPVOID l_pError = NULL;
if( pErrorBlob )
{
    l_pError = pErrorBlob->GetBufferPointer();
    // then cast to a char* to see it in the locals window
}
```

Related topics

[Rendering an Effect \(Direct3D 11\)](#)

Create an Effect (Direct3D 11)

2/22/2020 • 2 minutes to read • [Edit Online](#)

An effect is created by loading the compiled effect bytecode into the effects framework. Unlike Effects 10, the effect must be compiled before creating the effect. Effects loaded into memory can be created by calling [D3DX11CreateEffectFromMemory](#).

Related topics

[Rendering an Effect \(Direct3D 11\)](#)

Set Effect State (Direct3D 11)

2/22/2020 • 4 minutes to read • [Edit Online](#)

Some effect constants only need to be initialized. Once initialized, the effect state is set to the device for the entire render loop. Other variables need to be updated each time the render loop is called. The basic code for setting effect variables is shown below, for each of the types of variables.

An effect encapsulates all of the render state required to do a rendering pass. In terms of the API, there are three types of state encapsulated in an effect.

- [Constant State](#)
- [Shader State](#)
- [Texture State](#)

Constant State

First, declare variables in an effect using HLSL data types.

```
-----  
// Global variables  
-----  
float4 g_MaterialAmbientColor;      // Material's ambient color  
float4 g_MaterialDiffuseColor;      // Material's diffuse color  
int g_nNumLights;  
  
float3 g_LightDir[3];                // Light's direction in world space  
float4 g_LightDiffuse[3];            // Light's diffuse color  
float4 g_LightAmbient;              // Light's ambient color  
  
Texture2D g_MeshTexture;           // Color texture for mesh  
  
float    g_fTime;                  // App's time in seconds  
float4x4 g_mWorld;                // World matrix for object  
float4x4 g_mWorldViewProjection;   // World * View * Projection matrix
```

Second, declare variables in the application that can be set by the application, and will then update the effect variables.

```

D3DXMATRIX mWorldViewProjection;
D3DXVECTOR3 vLightDir[MAX_LIGHTS];
D3DXVECTOR4 vLightDiffuse[MAX_LIGHTS];
D3DXMATRIX mWorld;
D3DXMATRIX mView;
D3DXMATRIX mProj;

// Get the projection and view matrix from the camera class
mWorld = g_mCenterMesh * *g_Camera.GetWorldMatrix();
mProj = *g_Camera.GetProjMatrix();
mView = *g_Camera.GetViewMatrix();

OnD3D11CreateDevice()
{
    ...
    g_pLightDir = g_pEffect11->GetVariableByName( "g_LightDir" )->AsVector();
    g_pLightDiffuse = g_pEffect11->GetVariableByName( "g_LightDiffuse" )->AsVector();
    g_pmWorldViewProjection = g_pEffect11->GetVariableByName(
        "g_mWorldViewProjection" )->AsMatrix();
    g_pmWorld = g_pEffect11->GetVariableByName( "g_mWorld" )->AsMatrix();
    g_pfTime = g_pEffect11->GetVariableByName( "g_fTime" )->AsScalar();
    g_pMaterialAmbientColor = g_pEffect11->GetVariableByName("g_MaterialAmbientColor")->AsVector();
    g_pMaterialDiffuseColor = g_pEffect11->GetVariableByName(
        "g_MaterialDiffuseColor" )->AsVector();
    g_pnNumLights = g_pEffect11->GetVariableByName( "g_nNumLights" )->AsScalar();
}

```

Third, use the update methods to set the value of the variables in the effect variables.

```

OnD3D11FrameRender()
{
    ...
    g_pLightDir->SetRawValue( vLightDir, 0, sizeof(D3DXVECTOR3)*MAX_LIGHTS );
    g_pLightDiffuse->SetFloatVectorArray( (float*)vLightDiffuse, 0, MAX_LIGHTS );
    g_pmWorldViewProjection->SetMatrix( (float*)&mWorldViewProjection );
    g_pmWorld->SetMatrix( (float*)&mWorld );
    g_pfTime->SetFloat( (float)fTime );
    g_pnNumLights->SetInt( g_nNumActiveLights );
}

```

Two Ways to Get the State in an Effect Variable

There are two ways to get the state contained in an effect variable. Given an effect that has been loaded into memory.

One way is to get the sampler state from an [ID3DX11EffectVariable](#) that has been cast as a sampler interface.

```

D3D11_SAMPLER_DESC sampler_desc;
ID3D11EffectSamplerVariable* l_pD3D11EffectVariable = NULL;
if( g_pEffect11 )
{
    l_pD3D11EffectVariable = g_pEffect11->GetVariableByName( "MeshTextureSampler" )->AsSampler();
    if( l_pD3D11EffectVariable->IsValid() )
        hr = (l_pD3D11EffectVariable->GetBackingStore( 0,
            &sampler_desc );
}

```

The other way is to get the sampler state from an [ID3D11SamplerState](#).

```

ID3D11SamplerState* l_ppSamplerState = NULL;
D3D11_SAMPLER_DESC sampler_desc;
ID3D11EffectSamplerVariable* l_pD3D11EffectVariable = NULL;
if( g_pEffect11 )
{
    l_pD3D11EffectVariable = g_pEffect11->GetVariableByName( "MeshTextureSampler" )->AsSampler();
    if( l_pD3D11EffectVariable->IsValid )
    {
        hr = l_pD3D11EffectVariable->GetSampler( 0,
                                                &l_ppSamplerState );
        if( l_ppSamplerState )
            l_ppSamplerState->GetDesc( &sampler_desc );
    }
}

```

Shader State

Shader state is declared and assigned in an effect technique, within a pass.

```

VertexShader vsRenderScene = CompileShader( vs_4_0, RenderSceneVS( 1, true, true );
technique10 RenderSceneWithTexture1Light
{
    pass P0
    {
        SetVertexShader( vsRenderScene );
        SetGeometryShader( NULL );
        SetPixelShader( CompileShader( ps_4_0, RenderScenePS( true ) ) );
    }
}

```

This works just like it would if you were not using an effect. There are three calls, one for each type of shader (vertex, geometry, and pixel). The first one, SetVertexShader, calls [ID3D11DeviceContext::VSSetShader](#). CompileShader is a special effect function that takes the shader profile(vs_4_0) and the name of the vertex shader function (RenderVS). In other words, each of these CompileShader calls compiles their associated shader function and returns a pointer to the compiled shader.

Note that not all shader state must be set. This pass does not include any SetHullShader or SetDomainShader calls, meaning that the currently bound hull and domain shaders will be unchanged.

Texture State

Texture state is a little more complex than setting a variable, because texture data is not simply read like a variable, it is sampled from a texture. Therefore, you must define the texture variable (just like a normal variable except it uses a texture type) and you must define the sampling conditions. Here is an example of a texture variable declaration and the corresponding sampling state declaration.

```

Texture2D g_MeshTexture;           // Color texture for mesh

SamplerState MeshTextureSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = Wrap;
    AddressV = Wrap;
};

```

Here is an example of setting a texture from an application. In this example, the texture is stored in the mesh data, that was loaded when the effect was created.

The first step is getting a pointer to the texture from the effect (from the mesh).

```
ID3D11EffectShaderResourceVariable* g_ptxDiffuse = NULL;  
  
// Obtain variables  
g_ptxDiffuse = g_pEffect11->GetVariableByName( "g_MeshTexture" )->AsShaderResource();
```

The second step is specifying a view for accessing the texture. The view defines a general way to access the data from the texture resource.

```
OnD3D11FrameRender()  
{  
    ID3D11ShaderResourceView* pDiffuseRV = NULL;  
  
    ...  
    pDiffuseRV = g_Mesh11.GetMaterial(pSubset->MaterialID)->pDiffuseRV11;  
    g_ptxDiffuse->SetResource( pDiffuseRV );  
    ...  
}
```

From the application perspective, unordered access views are handled similarly to shader resource views. However, in the effect pixel shader and compute shader functions, unordered access view data is read from/written to directly. You cannot sample from an unordered access view.

For more information about viewing resources, see [Resources](#).

Related topics

[Rendering an Effect \(Direct3D 11\)](#)

Apply a Technique (Direct3D 11)

2/22/2020 • 2 minutes to read • [Edit Online](#)

With the constants, textures, and shader state declared and initialized, the only thing left to do is to set the effect state in the device.

Set Non-Shader State in the Device

Some pipeline state is not set by an effect. For example clearing a render target prepares the render target for data. Before setting the effect state in the device, here is an example of clearing output buffers.

```
// Clear the render target and depth stencil
float ClearColor[4] = { 0.0f, 0.25f, 0.25f, 0.55f };
ID3D11RenderTargetView* pRTV = DXUTGetD3D11RenderTargetView();
pD3DDevice->ClearRenderTargetView( pRTV, ClearColor );
ID3D11DepthStencilView* pDSV = DXUTGetD3D11DepthStencilView();
pD3DDevice->ClearDepthStencilView( pDSV, D3D11_CLEAR_DEPTH, 1.0, 0 );
```

Set Effect State in the Device

Setting effect state is done by applying the effect state within the render loop. This is done from the outside in. That is, select a technique, and then set the state for each of the passes (depending on your desired result).

```
D3D11_TECHNIQUE_DESC techDesc;
pRenderTechnique->GetDesc( &techDesc );
for( UINT p = 0; p < techDesc.Passes; ++p )
{
    ....
    pRenderTechnique->GetPassByIndex( p )->Apply(0);
    pd3dDevice->DrawIndexed( (UINT)pSubset->IndexCount, 0,
                            (UINT)pSubset->VertexStart );
}
```

An effect doesn't render anything, it simply sets effect state to the device. The rendering code is called after the effect state updates device state. In this example, the DrawIndexed call performs the rendering.

Related topics

[Rendering an Effect \(Direct3D 11\)](#)

Cloning an Effect

2/22/2020 • 2 minutes to read • [Edit Online](#)

Cloning an effect creates a second, almost identical copy of the effect. See the following single qualifier for an explanation of why it is not exact. A second copy of an effect is useful when one wants to use the effects framework on multiple threads, since the effect runtime is not thread safe to maintain high performance.

Since device contexts are also non-thread-safe, different threads should pass different device contexts to the ID3DX11EffectPass::Apply method.

An effect can be cloned with the following syntax:

```
ID3DX11Effect* pClonedEffect = NULL;  
UINT Flags = D3DX11_EFFECT_CLONE_FORCE_NONSINGLE;  
HRESULT hr = pEffect->CloneEffect( Flags, &pClonedEffect );
```

In the above example, the cloned copy will encapsulate the same state as the original effect, regardless of what state the original effect is in. In particular:

1. If pEffect is optimized, pCloned Effect is optimized
2. If pEffect has some user-managed variables, pCloned Effect will have the same user-managed variables (see the single description below)
3. Any pending variable updates (until an Apply call updates device state) in pEffect will be pending in pClonedEffect

The following Direct3D 11 device objects are immutable or never updated by the effects framework, so the cloned effect will point to the same objects as the original effect:

1. State block objects (ID3D11BlendState, ID3D11RasterizerState, ID3D11DepthStencilState, ID3D11SamplerState)
2. Shaders
3. Class instances
4. Textures (not including texture buffers)
5. Unordered access views

The following Direct3D 11 device objects are both immutable and modified by the effect runtime (unless user-managed or single in a cloned effect); new copies of these objects are created when non-single:

1. Constant buffers
2. Texture Buffers

Single Constant Buffers and Texture Buffers

Note that this discussion applies to both constant buffers and textures, but constant buffers are assumed for ease of reading.

There may be cases where a constant buffer is only updated by one thread, but device state set by cloned effects will use this data. For example, the main effect may update the world and view matrices which are referenced from shaders in cloned effects which do not change the world and view matrices. In these cases, the cloned effects need to reference the current constant buffer instead of recreating one.

There are two ways to achieve this desired result:

1. Use ID3DX11EffectConstantBuffer::SetConstantBuffer on the cloned effect to make it user-managed
2. Mark the constant buffer as "single" in the HLSL code, forcing the effect runtime to treat it as user-managed after cloning

There are two differences between the two methods above. First, in method 1, a new ID3D11Buffer will be created and used before SetConstantBuffer is called. Also, after calling UndoSetConstantBuffer in the cloned effect, the variable in method 1 will point to the newly-created buffer (which effects will update on Apply) while the variable in method 2 will continue to point to the original buffer (not updating it on Apply).

See the following example in HLSL:

```
cbuffer ObjectData
{
    float4 Position;
};

single cbuffer ViewData
{
    float4x4 ViewMatrix;
};
```

While cloning, the cloned effect will create a new ID3D11Buffer for ObjectData, and fill its contents on Apply, but reference the original ID3D11Buffer for ViewData. The single qualifier can be ignored in the cloning process by setting the D3DX11_EFFECT_CLONE_FORCE_NONSINGLE flag.

Related topics

[Effects \(Direct3D 11\)](#)

Stream Out Syntax

2/22/2020 • 2 minutes to read • [Edit Online](#)

A geometry shader with stream out is declared with a particular syntax. This topic describes the syntax. In the effect runtime, this syntax will be converted to a call to

[ID3D11Device::CreateGeometryShaderWithStreamOutput](#).

Construct Syntax

```
[ StreamingShaderVar = ] ConstructGSWithSO( ShaderVar, "OutputDecl0" )
```

NAME	DESCRIPTION
StreamingShaderVar	Optional. An ASCII string that uniquely identifies the name of a geometry shader variable with stream out. This is optional because ConstructGSWithSO can be placed directly in a SetGeometryShader or BindInterfaces call.
ShaderVar	A geometry shader or vertex shader variable.
OutputDecl0	A string defining which shader outputs in stream 0 are streamed out. See below for syntax.

This is the syntax was defined in fx_4_0 files. Note that in gs_4_0 and vs_x shaders, there is only one stream of data. The resulting shader will output one stream to both the stream out unit and the rasterizer unit.

```
[ StreamingShaderVar = ] ConstructGSWithSO( ShaderVar, "OutputDecl0", "OutputDecl1", "OutputDecl2",
"OutputDecl3", RasterizedStream )
```

NAME	DESCRIPTION
StreamingShaderVar	Optional. An ASCII string that uniquely identifies the name of a geometry shader variable with stream out. This is optional because ConstructGSWithSO can be placed directly in a SetGeometryShader or BindInterfaces call.
ShaderVar	A geometry shader or vertex shader variable.
OutputDecl0	A string defining which shader outputs in stream 0 are streamed out. See below for syntax.
OutputDecl1	A string defining which shader outputs in stream 1 are streamed out. See below for syntax.
OutputDecl2	A string defining which shader outputs in stream 2 are streamed out. See below for syntax.

NAME	DESCRIPTION
OutputDecl3	A string defining which shader outputs in stream 3 are streamed out. See below for syntax.
RasterizedStream	An integer specifying which stream will be sent to the rasterizer.

Note that gs_5_0 shaders can define up to four streams of data. The resulting shader will output one stream to the stream out unit for each non-NULL output declaration and one stream the rasterizer unit.

Stream Out Declaration Syntax

```
" [ Buffer: ] Semantic[ SemanticIndex ] [ .Mask ]; [ ... ; ] ... [ ... ; ]"
```

NAME	DESCRIPTION
Buffer	Optional. An integer, $0 \leq \text{Buffer} < 4$, specifying which stream out buffer the value will go to.
Semantic	A string, along with SemanticIndex, specifying which value to output.
SemanticIndex	Optional. The index associated with Semantic.
Mask	Optional. A component mask, indicating which components of the value to output.

There is one special Semantic, labeled "\$SKIP" which indicates an empty semantic, leaving the corresponding memory in the stream out buffer untouched. The \$SKIP semantic cannot have a SemanticIndex, but can have a Mask.

The entire stream out declaration can be NULL.

Example

```

struct GSOutput
{
    int4 Pos : Position;
    int4 Color : Color;
    int4 Texcoord : Texcoord;
};

[maxvertexcount(1)]
void gsBase (inout PointStream<GSOutput> OutputStream, inout PointStream<GSOutput> OutputStream1)
{
    GSOutput output;
    output.Pos = int4(1,2,3,4);
    output.Color = int4(5,6,7,8);
    output.Texcoord = int4(9,10,11,12);
    OutputStream.Append(output);

    output.Pos = int4(1,2,3,4);
    output.Color = int4(5,6,7,8);
    output.Texcoord = int4(9,10,11,12);
    OutputStream1.Append(output);
}

GeometryShader pGSComp = CompileShader(gs_5_0, gsBase());
GeometryShader pGSwSO = ConstructGSWithSO(pGSComp, "0:Position.xy; 1:Position.zw; 2:Color.xy",
                                         "3:Texcoord.xyzw; 3:$SKIP.x;", NULL, NULL, 1);

// The following two passes perform the same operation
technique11 SOPoints
{
    pass
    {
        SetGeometryShader(ConstructGSWithSO(pGSComp, "0:Position.xy; 1:Position.zw; 2:Color.xy",
                                            "3:Texcoord.xyzw; 3:$SKIP.x;", NULL, NULL, 1));
    }
    pass
    {
        SetGeometryShader(pGSwSO);
    }
}

```

Related topics

[Effects \(Direct3D 11\)](#)

Differences Between Effects 10 and Effects 11

11/2/2020 • 4 minutes to read • [Edit Online](#)

This topic shows the differences between Effects 10 and Effects 11.

Device Contexts, Threading, and Cloning

The ID3D10Device interface has been split into two interfaces in Direct3D 11: ID3D11Device and ID3D11DeviceContext. You can create multiple ID3D11DeviceContexts to facilitate concurrent execution on multiple threads. Effects 11 extends this concept to the Effects framework.

The Effects 11 runtime is single-threaded. For this reason, you should not use a single ID3DX11Effect instance with multiple threads concurrently.

To use the Effects 11 runtime on multiple instances, you must create separate ID3DX11Effect instances. Because the ID3D11DeviceContext is also single-threaded, you should pass different ID3D11DeviceContext instances to each effect instance on Apply. You can use these separate device contexts to create command lists so that the rendering thread can apply them on the immediate device context.

The easiest way to create multiple effects that encapsulate the same functionality, for use on multiple threads, is to create one effect and then make cloned copies. Cloning has the following advantages over creating multiple copies from scratch:

1. The cloning routine is faster than the creation routine.
2. Cloned effects share created shaders, state blocks, and class instances (so they don't have to be recreated).
3. Cloned effects can share constant buffers.
4. Cloned effects begin with state that matches the current effect (variable values, whether or not it has been optimized).

See [Cloning an Effect](#) for more information.

Effect Pools and Groups

By far the most prevalent use of effect pools in Direct3D 10 was for grouping materials. Effect Pools have been removed from Effects 11 and groups have been added, which is a more efficient method of grouping materials.

An effect group is simply a set of techniques. See [Effect Group Syntax \(Direct3D 11\)](#) for more information.

Consider the following effect hierarchy with four child effects and one effect pool:

```
// Effect Pool
cbuffer A { ... }
PixelShader pPSGrass;
PixelShader pPSWater;

// Effect Child 1
#include "EffectPool.fx"
technique10 GrassMaterialLowSpec { ... }

// Effect Child 2
#include "EffectPool.fx"
technique10 GrassMaterialHighSpec { ... }

// Effect Child 3
#include "EffectPool.fx"
technique10 WaterMaterialLowSpec { ... }

// Effect Child 4
#include "EffectPool.fx"
technique10 WaterMaterialHighSpec { ... }
```

You can achieve the same functionality in Effects 11 by using groups:

```
cbuffer A { ... }
PixelShader pPSGrass;
PixelShader pPSWater;

fxgroup GrassMaterial
{
    technique10 LowSpec { ... }
    technique10 HighSpec { ... }
}

fxgroup WaterMaterial
{
    technique10 LowSpec { ... }
    technique10 HighSpec { ... }
}
```

New Shader Stages

There are three new shader stages in Direct3D 11: the hull shader, domain shader, and compute shader. Effects 11 handles these in a similar manner to vertex shaders, geometry shaders, and pixel shaders.

Three new variable types have been added to Effects 11:

- HullShader
- DomainShader
- ComputeShader

If you use these shaders in a technique, you must label that technique "technique11", and not "technique10". The compute shader cannot be set in the same pass as any other graphics state (other shaders, state blocks, or render targets).

New Texture Types

Direct3D 11 supports the following texture types:

- [AppendStructuredBuffer](#)
- [ByteAddressBuffer](#)
- [ConsumeStructuredBuffer](#)
- [StructuredBuffer](#)

Unordered Access Views

Effects 11 supports getting and setting the new unordered access view types. This works in a similar manner as textures.

Consider this Effects HLSL example:

```
RWTexture1D<float> myUAV;
```

You can set this variable in C++ as follows:

```
ID3D11UnorderedAccessView* pUAVTexture1D = NULL;
ID3DX11EffectUnorderedAccessViewVariable* pUAVVar;
pUAVVar = pEffect->GetVariableByName("myUAV")->AsUnorderedAccessView();
pUAVVar->SetUnorderedAccessView( pUAVTexture1D );
```

Direct3D 11 supports the following unordered access view types:

- RWBuffer

- RWByteAddressBuffer
- RWStructuredBuffer
- RWTexture1D
- RWTexture1DArray
- RWTexture2D
- RWTexture2DArray
- RWTexture3D

Interfaces and Class Instances

For interface and class syntax, see [Interfaces and Classes](#).

For using interfaces and classes in effects, see [Interfaces and Classes in Effects](#).

Addressable Stream Out

In Direct3D 10, geometry shaders could output one stream of data to the stream output unit and the rasterizer unit. In Direct3D11, geometry shaders can output up to four streams of data to the stream output unit, and at most one of those streams to the rasterizer unit. The **ConstructGSWithSO** intrinsic has been updated to reflect this new functionality.

See [Stream Out Syntax](#) for more information.

Setting and Unsetting Device State

In Effects 10, you could make constant buffers and texture buffers user-managed by using the **ID3D10EffectConstantBuffer::SetConstantBuffer** and **SetTextureBuffer** functions. After you call these functions, the Effects 10 runtime no longer manages the constant buffer or texture buffer and the user must fill the data by using the ID3D10Device interface.

In Effects 11, you can also make the state blocks (blend state, rasterizer state, depth-stencil state, and sampler state) user-managed by using the following calls:

- [ID3DX11EffectBlendVariable::SetBlendState](#)
- [ID3DX11EffectRasterizerVariable::SetRasterizerState](#)
- [ID3DX11EffectDepthStencilVariable::SetDepthStencilState](#)
- [ID3DX11EffectSamplerVariable::SetSampler](#)

After you call these functions, the Effects 11 runtime no longer manages the state block variables, and their values will remain unchanged. Note that because state blocks are immutable, the user must set a new state block to change the values.

You can also revert constant buffers, texture buffers, and state blocks to the non-user managed state. If you unset these variables, the Effects 11 runtime will continue to update them when necessary. You can use the following calls to unset user managed variables:

- [ID3DX11EffectConstantBuffer::UndoSetConstantBuffer](#)
- [ID3DX11EffectConstantBuffer::UndoSetTextureBuffer](#)
- [ID3DX11EffectBlendVariable::UndoSetBlendState](#)
- [ID3DX11EffectRasterizerVariable::UndoSetRasterizerState](#)
- [ID3DX11EffectDepthStencilVariable::UndoSetDepthStencilState](#)
- [ID3DX11EffectSamplerVariable::UndoSetSampler](#)

Effects Virtual Machine

The effects virtual machine, which evaluated complex expressions outside of functions, has been removed.

The following examples of complex expressions are not supported:

1. SetPixelShader(myPSArray(i * 3 + j));

2. SetPixelShader(myPSArray((float)i);
3. FILTER = i + 2;

The following examples of non-complex expressions are supported:

1. SetPixelShader(myPS);
2. SetPixelShader(myPS[i]);
3. SetPixelShader(myPS[ulIndex]); // ulIndex is a uint variable
4. FILTER = i;
5. FILTER = ANISOTROPIC;

These expressions could appear in state block expressions (such as FILTER) and pass expressions (such as SetPixelShader).

Source Availability and Location

Effects 10 was distributed in D3D10.dll. Effects 11 is distributed as source, with corresponding Visual Studio solutions to compile it. When you create effects-type applications, we recommend that you include the Effects 11 source directly in those applications.

You can get Effects 11 from [Effects for Direct3D 11 Update](#).

Related topics

[Effects \(Direct3D 11\)](#)

Migrating to Direct3D 11

11/2/2020 • 6 minutes to read • [Edit Online](#)

This section provides info for migrating to Direct3D 11 from an earlier version of Direct3D.

- [Direct3D 9 to Direct3D 11](#)
- [Direct3D 10 to Direct3D 11](#)
 - [Enumerations and Defines](#)
 - [Structures](#)
 - [Interfaces](#)
 - [Other Related Technologies](#)
- [Direct3D 10.1 to Direct3D 11](#)
 - [Enumerations and Defines](#)
 - [Structures](#)
 - [Interfaces](#)
- [New Features for Direct3D 11](#)
- [New Features for DirectX 11.1](#)
- [New Features for DirectX 11.2](#)
- [Related topics](#)

Direct3D 9 to Direct3D 11

The Direct3D 11 API builds on the infrastructural improvements made in Direct3D 10 and 10.1. Porting from Direct3D 9 to Direct3D 11 is similar to moving from Direct3D 9 to Direct3D 10. The following are the key challenges in this effort.

- Removal of all fixed function pipeline usage in favor of programmable shaders authored exclusively in HLSL (compiled via D3DCompiler instead of D3DX9).
- State management based on immutable objects rather than individual state toggles.
- Updating to comply with strict linkage requirements of vertex buffer input layouts and shader signatures.
- Associating shader resource views with all texture resources.
- Mapping all image content to a DXGI_FORMAT, including the removal of all 16-bit color formats (5/5/5/1, 5/6/5, 4/4/4/4), removal of all 24-bit color formats (8/8/8), and strict RGB color ordering.
- Breaking up global constant state usage into several small, more efficiently updated constant buffers.

For more information about moving from Direct3D 9 to Direct3D 10, see [Direct3D 9 to Direct3D 10 Considerations](#).

Direct3D 10 to Direct3D 11

Converting programs written to use the Direct3D 10 or 10.1 API is a straight-forward process as Direct3D 11 is an extension of the existing API. With only one minor exception (noted below - monochrome text filtering), all methods and functionality in Direct3D 10/10.1 is available in Direct3D 11. The outline below describes the differences between the two APIs to aid in updating existing code. The key differences here include:

- Rendering operations (Draw, state, etc.) are no longer part of the Device interface, but are instead part of the new DeviceContext interface along with resource Map/Unmap and device query methods.
- Direct3D 11 includes all enhancements and changes made between Direct3D 10.0 and 10.1

Enumerations and Defines

DIRECT3D 10	DIRECT3D 11
DXGI_FORMAT	DXGI_FORMAT Several new DXGI formats were defined.
D3D10_CREATE_DEVICE_SWITCH_TO_REF	D3D11_CREATE_DEVICE_SWITCH_TO_REF The switch-to-ref functionality is not supported by Direct3D 11.
D3D10_DRIVER_TYPE	D3D_DRIVER_TYPE Note that the enumeration identifiers in D3D_DRIVER_TYPE were redefined from the identifiers in D3D10_DRIVER_TYPE . Therefore, you should be sure to use the enumeration identifiers instead of literal numbers. D3D_DRIVER_TYPE is defined in D3DCommon.h.
D3D10_RESOURCE_MISC_FLAG	D3D11_RESOURCE_MISC_FLAG Note that many of these flags were redefined, so be sure to use enumeration identifiers instead of literal numbers.
D3D10_FILTER	D3D11_FILTER Note that text filtering D3D10_FILTER_TEXT_1BIT was removed from Direct3D 11. See DirectWrite.
D3D11_COUNTER	D3D11_COUNTER Note that the vendor-neutral counters were removed for Direct3D 11 as they were rarely supported.
D3D10_x	D3D11_x Many enumerations and defines are the same, have larger limits, or have additional values.

Structures

DIRECT3D 10	DIRECT3D 11
D3D10_SO_DECLARATION_ENTRY	D3D11_SO_DECLARATION_ENTRY Adds Stream.
D3D10_BLEND_DESC	D3D11_BLEND_DESC Note this structure changed significantly from 10 to 10.1 to provide per render target blend state
D3D10_BUFFER_DESC	D3D11_BUFFER_DESC Adds a StructureByteStride for use with Compute Shader resources
D3D10_SHADER_RESOURCE_VIEW_DESC	D3D11_SHADER_RESOURCE_VIEW_DESC Note this structure had additional union members added for 10.1
D3D10_DEPTH_STENCIL_VIEW_DESC	D3D11_DEPTH_STENCIL_VIEW_DESC Has a new Flags member.
D3D10_QUERY_DATA_PIPELINE_STATISTICS	D3D11_QUERY_DATA_PIPELINE_STATISTICS Adds several new shader stage counters.
D3D10_x	D3D11_x Many structures are identical between the two APIs.

Interfaces

DIRECT3D 10	DIRECT3D 11
ID3D10Device	<p>ID3D11Device and ID3D11DeviceContext</p> <p>The device interface was split into these two parts. For quick porting you can make use of ID3D11Device::GetImmediateContext.</p> <p>"ID3D10Device::GetTextFilterSize" and "SetTextFilerSize" methods no longer exist. See DirectWrite.</p> <p>Create*Shader takes an additional optional parameter for ID3D11ClassLinkage.</p> <p>*SetShader and *GetShader take additional optional parameters for ID3D11ClassInstance(s).</p> <p>CreateGeometryShaderWithStreamOutput takes an array and count for multiple output stream strides.</p> <p>The limit for the NumEntries parameter of CreateGeometryShaderWithStreamOutput has increased to D3D11_SO_STREAM_COUNT * D3D11_SO_OUTPUT_COMPONENT_COUNT in Direct3D 11.</p>
ID3D10Buffer	ID3D11Buffer
ID3D10SwitchToRef	ID3D11SwitchToRef Switch-to-ref functionality is not supported in Direct3D 11.
ID3D10Texture1D	ID3D11Texture1D
ID3D10Texture2D	ID3D11Texture2D
ID3D10Texture3D	ID3D11Texture3D The Map and Unmap methods were moved to ID3D11DeviceContext , and all Map methods use D3D11_MAPPED_SUBRESOURCE instead of a void**.
ID3D10Asynchronous	ID3D11Asynchronous Begin, End, and GetData were moved to ID3D11DeviceContext .
ID3D10x	ID3D11x Many interfaces are identical between the two APIs.

Other Related Technologies

10/10.1 SOLUTION	11 SOLUTION
HLSL Complier (D3D10Compile*, D3DX10Compile*) and shader reflection APIs	<p>D3DCompiler (see D3DCompiler.h)</p> <div style="border: 1px solid #ccc; padding: 5px;"> <p>[!Note]</p> <p>For Windows Store apps, the D3DCompiler APIs are supported only for development, not deployment.</p> </div>
Effects 10	<p>Effects 11 is available as shared source online.</p> <div style="border: 1px solid #ccc; padding: 5px;"> <p>[!Note]</p> <p>This solution is not suited to Windows Store apps because it requires the D3DCompiler APIs at runtime (deployment).</p> </div>

10/10.1 SOLUTION	11 SOLUTION
D3DX9/D3DX10 Math	DirectXMath
D3DX10	D3DX11 in the legacy DirectX SDK DirectXTex , DirectXTK , and DirectXMesh offer alternatives to many technologies in the legacy D3DX10 and D3DX11 libraries. Direct2D and DirectWrite offer high-quality support for rendering styled lines and fonts.

For info about the legacy DirectX SDK, see [Where is the DirectX SDK?](#).

Direct3D 10.1 to Direct3D 11

Direct3D 10.1 is an extension of the Direct3D 10 interface, and all features of Direct3D 10.1 are available in Direct3D 11. Most of the porting from 10.1 to 11 is already addressed above moving from 10 to 11.

Enumerations and Defines

DIRECT3D 10.1	DIRECT3D 11
D3D10_FEATURE_LEVEL1	D3D_FEATURE_LEVEL Identical but defined in D3DCommon.h plus the addition of D3D_FEATURE_LEVEL_11_0 for 11 class hardware along with D3D_FEATURE_LEVEL_9_1, D3D_FEATURE_LEVEL_9_2, and D3D_FEATURE_LEVEL_9_3 for 10 level 9 (D3D10_FEATURE_LEVEL_9_1, D3D10_FEATURE_LEVEL_9_2, and D3D10_FEATURE_LEVEL_9_3 were also added for 10.1 to d3d10_1.h)

Structures

DIRECT3D 10.1	DIRECT3D 11
D3D10_BLEND_DESC1	D3D11_BLEND_DESC The 11 version is identical to 10.1.
D3D10_SHADER_RESOURCE_VIEW_DESC1	D3D11_SHADER_RESOURCE_VIEW_DESC The 11 version is identical to 10.1.

Interfaces

DIRECT3D 10.1	DIRECT3D 11

DIRECT3D 10.1	DIRECT3D 11
ID3D10Device1	<p>ID3D11Device and ID3D11DeviceContext</p> <p>The device interface was split into these two parts. For quick porting you can make use of ID3D11Device::GetImmediateContext.</p> <p>" ID3D10Device::GetTextFilterSize" and "SetTextFilerSize" methods no longer exist. See DirectWrite.</p> <p>Create*Shader takes an additional optional parameter for ID3D11ClassLinkage.</p> <p>*SetShader and *GetShader take additional optional parameters for ID3D11ClassInstance(s).</p> <p>CreateGeometryShaderWithStreamOutput takes an array and count for multiple output stream strides.</p> <p>The limit for the NumEntries parameter of CreateGeometryShaderWithStreamOutput has increased to D3D11_SO_STREAM_COUNT * D3D11_SO_OUTPUT_COMPONENT_COUNT in Direct3D 11.</p>
ID3D10BlendState1	ID3D11BlendState
ID3D10ShaderResourceView1	ID3D11ShaderResourceView

New Features for Direct3D 11

Once your code is updated to use the Direct3D 11 API, there are numerous [new features](#) to consider.

- Multithreaded rendering through command lists and multiple contexts
- Implementing advanced algorithms using Compute Shader (using 4.0, 4.1, or 5.0 shader profiles)
- New 11 class hardware features:
 - HLSL Shader Model 5.0
 - Dynamic Shader Linkage
 - Tessellation through Hull and Domain shaders
 - New block compression formats: BC6H for HDR images, BC7 for higher-fidelity standard images
- Utilizing [10level9 technology](#) for rendering on many Shader Model 2.0 and Shader Model 3.0 devices through the Direct3D 11 API for lower-end video hardware support on Windows Vista and Windows 7.
- Leveraging the WARP software rendering device.

New Features for DirectX 11.1

Windows 8 includes further DirectX graphics enhancements to consider when you implement your DirectX graphics code, which include [Direct3D 11.1](#), [DXGI 1.2](#), [Windows Display Driver Model \(WDDM\) 1.2](#), [feature level 11.1 hardware](#), Direct2D device contexts, and other improvements.

Partial support for [Direct3D 11.1](#) is available on Windows 7 as well via the [Platform Update for Windows 7](#), which is available through the [Platform Update for Windows 7](#).

New Features for DirectX 11.2

The Windows 8.1 includes [Direct3D 11.2](#), [DXGI 1.3](#), and other improvements.

Related topics

[Programming Guide for Direct3D 11](#)

Effects (Direct3D 11)

Direct3D Video Interfaces

11/2/2020 • 2 minutes to read • [Edit Online](#)

This topic lists the Direct3D video interfaces that are available in Direct3D 9Ex and that are supported on Windows 7 and later versions of Windows client operating systems and on Windows Server 2008 R2 and later versions of Windows server operating systems. You can use these interfaces and their methods to obtain the video content protection capabilities of the graphics driver and the overlay hardware capabilities of a Direct3D device. You can also use these interfaces and their methods to protect video content. These interfaces and their methods are defined in D3d9.h and described in the [Microsoft Media Foundation](#) section.

INTERFACE	DESCRIPTION
IDirect3D9ExOverlayExtension	Queries the overlay hardware capabilities of a Direct3D device.
IDirect3DAuthenticatedChannel9	Provides a communication channel with the graphics driver or the Direct3D runtime.
IDirect3DCryptoSession9	Represents a cryptographic session that is used to access a protected surface.
IDirect3DDevice9Video	Enables an application to use content protection and encryption services that are implemented by a graphics driver.

Related topics

[Effects \(Direct3D 11\)](#)

[Programming Guide for Direct3D 11](#)

Direct3D 11, utilities, and effects reference

2/22/2020 • 2 minutes to read • [Edit Online](#)

This section contains the reference pages for Direct3D 11-based graphics programming.

In this section

TOPIC	DESCRIPTION
Direct3D 11 Reference	The Direct3D 11 API is described in this section.
D3DCSX 11 Reference	This section contains reference information about the D3DCSX utility library, which you can use with a compute shader.
D3DX 11 Reference	The D3DX 11 API is described in this section.
Effects 11 Reference	Use Effects 11 source to build your effects-type application. The Effects 11 source is available at Effects for Direct3D 11 Update . The Effects 11 API is described in this section.

Related topics

[Direct3D 11 Graphics](#)

Direct3D 11 Reference

2/22/2020 • 2 minutes to read • [Edit Online](#)

The Direct3D 11 API is described in this section.

In this section

TOPIC	DESCRIPTION
Core Reference	The Direct3D API defines several core API elements.
Layer Reference	The Direct3D API defines several layer API elements.
Resource Reference	The Direct3D API defines several API elements to help you create and manage resources.
Shader Reference	The Direct3D API defines several API elements to help you create and manage programmable shaders. Shaders are executable programs that are programmed exclusively using HLSL.
10Level9 Reference	This section specifies the differences between each 10Level9 feature level and the D3D_FEATURE_LEVEL_11_0 and higher feature level for the ID3D11Device and ID3D11DeviceContext methods.
Direct3D 11 Return Codes	The return codes from API functions.
Common Version Reference	The Direct3D API defines several API elements that are common to the Direct3D 12, Direct3D 11, Direct3D 10, and Direct3D 10.1. You can use these API elements in your code for any of these Direct3D versions. These API elements are known as version neutral.
CD3D11 Helper Structures	Direct3D 11 defines several helper structures that you can use to create Direct3D structures. These helper structures behave like C++ classes.

Related topics

[Direct3D 11 Reference](#)

[Direct3D 11 Graphics](#)

Direct3D 11 core reference

2/4/2021 • 2 minutes to read • [Edit Online](#)

The Direct3D API defines several core API elements.

In this section

TOPIC	DESCRIPTION
Core Interfaces	This section contains information about the core interfaces.
Core Functions	This section contains information about the core functions.
Core Structures	This section contains information about the core structures.
Core Enumerations	This section contains information about the core enumerations.

Related topics

[Direct3D 11 Reference](#)

Direct3D 11 core interfaces

11/2/2020 • 3 minutes to read • [Edit Online](#)

This section contains information about the core interfaces.

In this section

TOPIC	DESCRIPTION
ID3D11Asynchronous	This interface encapsulates methods for retrieving data from the GPU asynchronously.
ID3D11BlendState	The blend-state interface holds a description for blending state that you can bind to the output-merger stage .
ID3D11BlendState1	The blend-state interface holds a description for blending state that you can bind to the output-merger stage . This blend-state interface supports logical operations as well as blending operations.
ID3D11CommandList	The ID3D11CommandList interface encapsulates a list of graphics commands for play back.
ID3D11Counter	This interface encapsulates methods for measuring GPU performance.
ID3D11DepthStencilState	The depth-stencil-state interface holds a description for depth-stencil state that you can bind to the output-merger stage .
ID3D11Device	The device interface represents a virtual adapter; it is used to create resources.
ID3D11Device1	The device interface represents a virtual adapter; it is used to create resources. ID3D11Device1 adds new methods to those in ID3D11Device .
ID3D11Device2	The device interface represents a virtual adapter; it is used to create resources. ID3D11Device2 adds new methods to those in ID3D11Device1 .
ID3D11Device3	The device interface represents a virtual adapter; it is used to create resources. ID3D11Device3 adds new methods to those in ID3D11Device2 .
ID3D11Device4	The device interface represents a virtual adapter; it is used to create resources. ID3D11Device4 adds new methods to those in ID3D11Device3 , such as RegisterDeviceRemovedEvent and UnregisterDeviceRemoved .

TOPIC	DESCRIPTION
ID3D11Device5	The device interface represents a virtual adapter; it is used to create resources. ID3D11Device5 adds new methods to those in ID3D11Device4 .
ID3D11DeviceChild	A device-child interface accesses data used by a device.
ID3D11DeviceContext	<p>The ID3D11DeviceContext interface represents a device context which generates rendering commands.</p> <div data-bbox="817 482 1420 707" style="border: 1px solid black; padding: 5px;"> <p>[!Note] The latest version of this interface is ID3D11DeviceContext4 introduced in the Windows 10 Creators Update. Applications targeting Windows 10 Creators Update should use the ID3D11DeviceContext4 interface instead of ID3D11Device.</p> </div>
ID3D11DeviceContext1	The device context interface represents a device context; it is used to render commands. ID3D11DeviceContext1 adds new methods to those in ID3D11DeviceContext .
ID3D11DeviceContext2	The device context interface represents a device context; it is used to render commands. ID3D11DeviceContext2 adds new methods to those in ID3D11DeviceContext1 .
ID3D11DeviceContext3	The device context interface represents a device context; it is used to render commands. ID3D11DeviceContext3 adds new methods to those in ID3D11DeviceContext2 .
ID3D11DeviceContext4	The device context interface represents a device context; it is used to render commands. ID3D11DeviceContext4 adds new methods to those in ID3D11DeviceContext3 .
ID3DDeviceContextState	The ID3DDeviceContextState interface represents a context state object, which holds state and behavior information about a Microsoft Direct3D device.
ID3D11Fence	Represents a fence, an object used for synchronization of the CPU and one or more GPUs.
ID3D11InputLayout	An input-layout interface holds a definition of how to feed vertex data that is laid out in memory into the input-assembler stage of the graphics pipeline .
ID3D11Multithread	Provides threading protection for critical sections of a multi-threaded application.
ID3D11Predicate	A predicate interface determines whether geometry should be processed depending on the results of a previous draw call.
ID3D11Query	A query interface queries information from the GPU.

TOPIC	DESCRIPTION
ID3D11Query1	Represents a query object for querying information from the graphics processing unit (GPU).
ID3D11RasterizerState	The rasterizer-state interface holds a description for rasterizer state that you can bind to the rasterizer stage .
ID3D11RasterizerState1	The rasterizer-state interface holds a description for rasterizer state that you can bind to the rasterizer stage . This rasterizer-state interface supports forced sample count.
ID3D11RasterizerState2	The rasterizer-state interface holds a description for rasterizer state that you can bind to the rasterizer stage . This rasterizer-state interface supports forced sample count and conservative rasterization mode.
ID3D11SamplerState	The sampler-state interface holds a description for sampler state that you can bind to any shader stage of the pipeline for reference by texture sample operations.

Direct3D 11 implements interfaces for:

- [Resources](#)
- [Shaders](#)

Related topics

[Core Reference](#)

Direct3D 11 core functions

2/22/2020 • 2 minutes to read • [Edit Online](#)

This section contains information about the core functions.

In this section

TOPIC	DESCRIPTION
D3D11CreateDevice	Creates a device that represents the display adapter.
D3D11CreateDeviceAndSwapChain	Creates a device that represents the display adapter and a swap chain used for rendering.

Related topics

[Core Reference](#)

Direct3D 11 core structures

11/2/2020 • 3 minutes to read • [Edit Online](#)

This section contains information about the core structures.

In this section

TOPIC	DESCRIPTION
D3D11_BLEND_DESC	Describes the blend state that you use in a call to ID3D11Device::CreateBlendState to create a blend-state object.
D3D11_BLEND_DESC1	<div style="border: 1px solid black; padding: 5px;"><p>[!Note] This structure is supported by the Direct3D 11.1 runtime, which is available on Windows 8 and later operating systems.</p></div>
	Describes the blend state that you use in a call to ID3D11Device1::CreateBlendState1 to create a blend-state object.
D3D11_BOX	Defines a 3D box.
D3D11_COUNTER_DESC	Describes a counter.
D3D11_COUNTER_INFO	Information about the video card's performance counter capabilities.
D3D11_DEPTH_STENCIL_DESC	Describes depth-stencil state.
D3D11_DEPTH_STENCILOP_DESC	Stencil operations that can be performed based on the results of stencil test.
D3D11_DRAW_INDEXED_INSTANCED_INDIRECT_ARGS	Arguments for draw indexed instanced indirect.
D3D11_DRAW_INSTANCED_INDIRECT_ARGS	Arguments for draw instanced indirect.
D3D11_FEATURE_DATA_ARCHITECTURE_INFO	<div style="border: 1px solid black; padding: 5px;"><p>[!Note] This structure is supported by the Direct3D 11.1 runtime, which is available on Windows 8 and later operating systems.</p></div>
	Describes information about Direct3D 11.1 adapter architecture.

Topic	Description
D3D11_FEATURE_DATA_D3D9_OPTIONS	<p data-bbox="826 204 917 233">[!Note]</p> <p data-bbox="826 238 1342 327">This structure is supported by the Direct3D 11.1 runtime, which is available on Windows 8 and later operating systems.</p> <p data-bbox="810 406 1414 467">Describes Direct3D 9 feature options in the current graphics driver.</p>
D3D11_FEATURE_DATA_D3D9_OPTIONS1	Describes Direct3D 9 feature options in the current graphics driver.
D3D11_FEATURE_DATA_D3D9_SHADOW_SUPPORT	<p data-bbox="826 669 917 698">[!Note]</p> <p data-bbox="826 702 1342 792">This structure is supported by the Direct3D 11.1 runtime, which is available on Windows 8 and later operating systems.</p> <p data-bbox="810 871 1334 932">Describes Direct3D 9 shadow support in the current graphics driver.</p>
D3D11_FEATURE_DATA_D3D9_SIMPLE_INSTANCING_SUPPORT	Describes whether simple instancing is supported.
D3D11_FEATURE_DATA_D3D10_X_HARDWARE_OPTIONS	Describes compute shader and raw and structured buffer support in the current graphics driver.
D3D11_FEATURE_DATA_D3D11_OPTIONS	<p data-bbox="826 1244 917 1273">[!Note]</p> <p data-bbox="826 1277 1342 1367">This structure is supported by the Direct3D 11.1 runtime, which is available on Windows 8 and later operating systems.</p> <p data-bbox="810 1446 1358 1507">Describes Direct3D 11.1 feature options in the current graphics driver.</p>
D3D11_FEATURE_DATA_D3D11_OPTIONS1	Describes Direct3D 11.2 feature options in the current graphics driver.
D3D11_FEATURE_DATA_D3D11_OPTIONS2	Describes Direct3D 11.3 feature options in the current graphics driver.
D3D11_FEATURE_DATA_D3D11_OPTIONS3	Describes Direct3D 11.3 feature options in the current graphics driver.
D3D11_FEATURE_DATA_D3D11_OPTIONS4	Describes Direct3D 11.4 feature options in the current graphics driver.
D3D11_FEATURE_DATA_D3D11_OPTIONS5	Describes the level of support for shared resources in the current graphics driver.

TOPIC	DESCRIPTION
D3D11_FEATURE_DATA_DOUBLES	Describes double data type support in the current graphics driver.
D3D11_FEATURE_DATA_FORMAT_SUPPORT	Describes which resources are supported by the current graphics driver for a given format.
D3D11_FEATURE_DATA_FORMAT_SUPPORT2	Describes which unordered resource options are supported by the current graphics driver for a given format.
D3D11_FEATURE_DATA_GPU_VIRTUAL_ADDRESS_SUPPORT	Describes feature data GPU virtual address support, including maximum address bits per resource and per process.
D3D11_FEATURE_DATA_MARKER_SUPPORT	Describes whether a GPU profiling technique is supported.
D3D11_FEATURE_DATA_SHADER_CACHE	Describes the level of shader caching supported in the current graphics driver.
D3D11_FEATURE_DATA_SHADER_MIN_PRECISION_SUPPORT	<p>[!Note] This structure is supported by the Direct3D 11.1 runtime, which is available on Windows 8 and later operating systems.</p> <p>Describes precision support options for shaders in the current graphics driver.</p>
D3D11_FEATURE_DATA_THREADING	Describes the multi-threading features that are supported by the current graphics driver.
D3D11_INPUT_ELEMENT_DESC	A description of a single element for the input-assembler stage.
D3D11_QUERY_DATA_PIPELINE_STATISTICS	Query information about graphics-pipeline activity in between calls to ID3D11DeviceContext::Begin and ID3D11DeviceContext::End .
D3D11_QUERY_DATA_SO_STATISTICS	Query information about the amount of data streamed out to the stream-output buffers in between ID3D11DeviceContext::Begin and ID3D11DeviceContext::End .
D3D11_QUERY_DATA_TIMESTAMP_DISJOINT	Query information about the reliability of a timestamp query.
D3D11_QUERY_DESC	Describes a query.
D3D11_QUERY_DESC1	Describes a query.
D3D11_RASTERIZER_DESC	Describes rasterizer state.

TOPIC	DESCRIPTION
D3D11_RASTERIZER_DESC1	<p>[!Note] This structure is supported by the Direct3D 11.1 runtime, which is available on Windows 8 and later operating systems.</p>
	Describes rasterizer state.
D3D11_RASTERIZER_DESC2	Describes rasterizer state.
D3D11_RECT	D3D11_RECT is declared as follows:
D3D11_RENDER_TARGET_BLEND_DESC	Describes the blend state for a render target.
D3D11_RENDER_TARGET_BLEND_DESC1	<p>[!Note] This structure is supported by the Direct3D 11.1 runtime, which is available on Windows 8 and later operating systems.</p>
	Describes the blend state for a render target.
D3D11_SAMPLER_DESC	Describes a sampler state.
D3D11_SO_DECLARATION_ENTRY	Description of a vertex element in a vertex buffer in an output slot.
D3D11_VIEWPORT	Defines the dimensions of a viewport.

In addition, there's a 2D rectangle structure defined in D3D11.h.

```
typedef RECT D3D11_RECT;
```

For documentation, see [RECT in Windows GDI](#).

Related topics

[Core Reference](#)

D3D11_RECT

11/2/2020 • 2 minutes to read • [Edit Online](#)

D3D11_RECT is declared as follows:

```
typedef RECT D3D11_RECT;
```

For more information about this GDI rectangle structure, see [RECT](#).

Remarks

This structure is used for scissor rectangles by [ID3D11DeviceContext::RSGetScissorRects](#) and [ID3D11DeviceContext::RSSetScissorRects](#).

Requirements

Header	D3D11.h
Library	D3D11.lib

See also

[Core Structures](#)

Direct3D 11 core enumerations

11/2/2020 • 2 minutes to read • [Edit Online](#)

This section contains information about the core enumerations.

In this section

TOPIC	DESCRIPTION
D3D11_ASYNC_GETDATA_FLAG	Optional flags that control the behavior of ID3D11DeviceContext::GetData .
D3D11_BLEND	Blend factors, which modulate values for the pixel shader and render target.
D3D11_BLEND_OP	RGB or alpha blending operation.
D3D11_CLEAR_FLAG	Specifies the parts of the depth stencil to clear.
D3D11_COLOR_WRITE_ENABLE	Identify which components of each pixel of a render target are writable during blending.
D3D11_COMPARISON_FUNC	Comparison options.
D3D11_CONSERVATIVE_RASTERIZATION_MODE	Identifies whether conservative rasterization is on or off.
D3D11_CONSERVATIVE_RASTERIZATION_TIER	Specifies if the hardware and driver support conservative rasterization and at what tier level.
D3D11_CONTEXT_TYPE	Specifies the context in which a query occurs.
D3D11_COPY_FLAGS	<p>[!Note]</p> <p>This enumeration is supported by the Direct3D 11.1 runtime, which is available on Windows 8 and later operating systems.</p> <p>Specifies how to handle the existing contents of a resource during a copy or update operation of a region within that resource.</p>
D3D11_COUNTER	Options for performance counters.
D3D11_COUNTER_TYPE	Data type of a performance counter.
D3D11_CREATE_DEVICE_FLAG	Describes parameters that are used to create a device.
D3D11_1_CREATE_DEVICE_CONTEXT_STATE_FLAG	Describes flags that are used to create a device context state object (ID3DDeviceContextState) with the ID3D11Device1::CreateDeviceContextState method.

TOPIC	DESCRIPTION
D3D11_CULL_MODE	Indicates triangles facing a particular direction are not drawn.
D3D11_DEPTH_WRITE_MASK	Identify the portion of a depth-stencil buffer for writing depth data.
D3D11_DEVICE_CONTEXT_TYPE	Device context options.
D3D11_FEATURE	Direct3D 11 feature options.
D3D11_FENCE_FLAG	Specifies fence options.
D3D11_FILL_MODE	Determines the fill mode to use when rendering triangles.
D3D11_FILTER	Filtering options during texture sampling.
D3D11_FILTER_TYPE	Types of magnification or minification sampler filters.
D3D11_FILTER_REDUCTION_TYPE	Specifies the type of sampler filter reduction.
D3D11_FORMAT_SUPPORT	Which resources are supported for a given format and given device (see ID3D11Device::CheckFormatSupport and ID3D11Device::CheckFeatureSupport).
D3D11_FORMAT_SUPPORT2	Unordered resource support options for a compute shader resource (see ID3D11Device::CheckFeatureSupport).
D3D11_INPUT_CLASSIFICATION	Type of data contained in an input slot.
D3D11_LOGIC_OP	<p>[!Note] This enumeration is supported by the Direct3D 11.1 runtime, which is available on Windows 8 and later operating systems.</p> <p>Specifies logical operations to configure for a render target.</p>
D3D11_PRIMITIVE	Indicates how the pipeline interprets geometry or hull shader input primitives.
D3D11_PRIMITIVE_TOPOLOGY	How the pipeline interprets vertex data that is bound to the input-assembler stage. These primitive topology values determine how the vertex data is rendered on screen.
D3D11_QUERY	Query types.
D3D11_QUERY_MISC_FLAG	Flags that describe miscellaneous query behavior.
D3D11_RAISE_FLAG	Option(s) for raising an error to a non-continuable exception.

TOPIC	DESCRIPTION
D3D11_SHADER_CACHE_SUPPORT_FLAGS	Describes the level of support for shader caching in the current graphics driver.
D3D11_SHADER_MIN_PRECISION_SUPPORT	<p data-bbox="826 316 917 345">[!Note]</p> <p data-bbox="826 350 1355 440">This enumeration is supported by the Direct3D 11.1 runtime, which is available on Windows 8 and later operating systems.</p> <p data-bbox="810 512 1355 570">Values that specify minimum precision levels at shader stages.</p>
D3D11_SHARED_RESOURCE_TIER	Defines constants that specify a tier for shared resource support.
D3D11_STENCIL_OP	The stencil operations that can be performed during depth-stencil testing.
D3D11_TEXTURE_ADDRESS_MODE	Identify a technique for resolving texture coordinates that are outside of the boundaries of a texture.
D3D11_TEXTURECUBE_FACE	The different faces of a cube texture.
D3D11_TILED_RESOURCES_TIER	Indicates the tier level at which tiled resources are supported.

Related topics

[Core reference](#)

Layer Reference

2/4/2021 • 2 minutes to read • [Edit Online](#)

The Direct3D API defines several layer API elements.

In this section

TOPIC	DESCRIPTION
Layer Interfaces	This section contains information about the layer interfaces.
Layer Structures	This section contains information about the layer structures.
Layer Enumerations	This section contains information about the layer enumerations.

Related topics

[Direct3D 11 Reference](#)

Layer Interfaces

2/4/2021 • 2 minutes to read • [Edit Online](#)

This section contains information about the layer interfaces.

In this section

TOPIC	DESCRIPTION
ID3D11Debug	A debug interface controls debug settings, validates pipeline state and can only be used if the debug layer is turned on.
ID3D11InfoQueue	An information-queue interface stores, retrieves, and filters debug messages. The queue consists of a message queue, an optional storage filter stack, and a optional retrieval filter stack.
ID3D11RefDefaultTrackingOptions	The default tracking interface sets reference default tracking options.
ID3D11RefTrackingOptions	The tracking interface sets reference tracking options.
ID3D11SwitchToRef	<p>[!Note] The ID3D11SwitchToRef interface and its methods are not supported in Direct3D 11.</p>
ID3D11TracingDevice	The tracing device interface sets shader tracking information, which enables accurate logging and playback of shader execution.

Related topics

[Layer Reference](#)

Layer Structures

2/4/2021 • 2 minutes to read • [Edit Online](#)

This section contains information about the layer structures.

In this section

TOPIC	DESCRIPTION
D3D11_INFO_QUEUE_FILTER	Debug message filter; contains a lists of message types to allow or deny.
D3D11_INFO_QUEUE_FILTER_DESC	Allow or deny certain types of messages to pass through a filter.
D3D11_MESSAGE	A debug message in the Information Queue.

Related topics

[Layer Reference](#)

Layer Enumerations

2/4/2021 • 2 minutes to read • [Edit Online](#)

This section contains information about the layer enumerations.

In this section

TOPIC	DESCRIPTION
D3D11_MESSAGE_CATEGORY	Categories of debug messages. This will identify the category of a message when retrieving a message with ID3D11InfoQueue::GetMessage and when adding a message with ID3D11InfoQueue::AddMessage . When creating an info queue filter , these values can be used to allow or deny any categories of messages to pass through the storage and retrieval filters.
D3D11_MESSAGE_ID	Debug messages for setting up an info-queue filter (see D3D11_INFO_QUEUE_FILTER); use these messages to allow or deny message categories to pass through the storage and retrieval filters. These IDs are used by methods such as ID3D11InfoQueue::GetMessage or ID3D11InfoQueue::AddMessage .
D3D11_MESSAGE_SEVERITY	Debug message severity levels for an information queue.
D3D11_RLDO_FLAGS	Options for the amount of information to report about a device object's lifetime.
D3D11_SHADER_TRACKING_OPTIONS	Options that specify how to perform shader debug tracking.
D3D11_SHADER_TRACKING_RESOURCE_TYPE	Indicates which resource types to track.

Related topics

[Layer Reference](#)

Resource Reference (Direct3D 11 Graphics)

2/4/2021 • 2 minutes to read • [Edit Online](#)

The Direct3D API defines several API elements to help you create and manage resources.

In this section

TOPIC	DESCRIPTION
Resource Interfaces	<p>There are a number of interfaces for the two basic types of resources: buffers and textures. There are also interfaces for views of resources.</p> <p>An application uses a view to bind a resource to a pipeline stage. The view defines how the resource can be accessed during rendering. This section describes the view interfaces.</p>
Resource Functions	This section contains information about the resource functions.
Resource Structures	Structures are used to create and use resources.
Resource Enumerations	Enumerations are used to specify information about how resources are created and accessed during rendering.

Related topics

[Direct3D 11 Reference](#)

Resource Interfaces (Direct3D 11 Graphics)

2/4/2021 • 2 minutes to read • [Edit Online](#)

There are a number of interfaces for the two basic types of resources: buffers and textures. There are also interfaces for views of resources.

An application uses a view to bind a resource to a pipeline stage. The view defines how the resource can be accessed during rendering. This section describes the view interfaces.

In this section

TOPIC	DESCRIPTION
ID3D11Buffer	A buffer interface accesses a buffer resource, which is unstructured memory. Buffers typically store vertex or index data.
ID3D11DepthStencilView	A depth-stencil-view interface accesses a texture resource during depth-stencil testing.
ID3D11RenderTargetView	A render-target-view interface identifies the render-target subresources that can be accessed during rendering.
ID3D11RenderTargetView1	A render-target-view interface represents the render-target subresources that can be accessed during rendering.
ID3D11Resource	A resource interface provides common actions on all resources.
ID3D11ShaderResourceView	A shader-resource-view interface specifies the subresources a shader can access during rendering. Examples of shader resources include a constant buffer, a texture buffer, and a texture.
ID3D11ShaderResourceView1	A shader-resource-view interface represents the subresources a shader can access during rendering. Examples of shader resources include a constant buffer, a texture buffer, and a texture.
ID3D11Texture1D	A 1D texture interface accesses texel data, which is structured memory.
ID3D11Texture2D	A 2D texture interface manages texel data, which is structured memory.
ID3D11Texture2D1	A 2D texture interface represents texel data, which is structured memory.
ID3D11Texture3D	A 3D texture interface accesses texel data, which is structured memory.

TOPIC	DESCRIPTION
ID3D11Texture3D1	A 3D texture interface represents texel data, which is structured memory.
ID3D11UnorderedAccessView	A view interface specifies the parts of a resource the pipeline can access during rendering.
ID3D11UnorderedAccessView1	An unordered-access-view interface represents the parts of a resource the pipeline can access during rendering.
ID3D11View	A view interface specifies the parts of a resource the pipeline can access during rendering.

Related topics

[Resource Reference](#)

Resource Functions (Direct3D 11 Graphics)

2/4/2021 • 2 minutes to read • [Edit Online](#)

This section contains information about the resource functions.

In this section

TOPIC	DESCRIPTION
D3D11CalcSubresource	Calculates a subresource index for a texture.

Related topics

[Resource Reference](#)

Resource Structures (Direct3D 11 Graphics)

2/4/2021 • 3 minutes to read • [Edit Online](#)

Structures are used to create and use resources.

In this section

TOPIC	DESCRIPTION
D3D11_BUFFER_DESC	Describes a buffer resource.
D3D11_BUFFER_RTV	Specifies the elements in a buffer resource to use in a render-target view.
D3D11_BUFFER_SRV	Specifies the elements in a buffer resource to use in a shader-resource view.
D3D11_BUFFER_UAV	Describes the elements in a buffer to use in a unordered-access view.
D3D11_BUFFEREX_SRV	Describes the elements in a raw buffer resource to use in a shader-resource view.
D3D11_DEPTH_STENCIL_VIEW_DESC	Specifies the subresources of a texture that are accessible from a depth-stencil view.
D3D11_MAPPED_SUBRESOURCE	Provides access to subresource data.
D3D11_PACKED_MIP_DESC	Describes the tile structure of a tiled resource with mipmaps.
D3D11_RENDER_TARGET_VIEW_DESC	Specifies the subresources from a resource that are accessible using a render-target view.
D3D11_RENDER_TARGET_VIEW_DESC1	Describes the subresources from a resource that are accessible using a render-target view.
D3D11_SHADER_RESOURCE_VIEW_DESC	Describes a shader-resource view.
D3D11_SHADER_RESOURCE_VIEW_DESC1	Describes a shader-resource view.
D3D11_SUBRESOURCE_DATA	Specifies data for initializing a subresource.
D3D11_SUBRESOURCE_TILING	Describes a tiled subresource volume.
D3D11_TEX1D_ARRAY_DSV	Specifies the subresources from an array of 1D textures to use in a depth-stencil view.
D3D11_TEX1D_ARRAY_RTV	Specifies the subresources from an array of 1D textures to use in a render-target view.

Topic	Description
D3D11_TEX1D_ARRAY_SRV	Specifies the subresources from an array of 1D textures to use in a shader-resource view.
D3D11_TEX1D_ARRAY_UAV	Describes an array of unordered-access 1D texture resources.
D3D11_TEX1D_DSV	Specifies the subresource from a 1D texture that is accessible to a depth-stencil view.
D3D11_TEX1D_RTV	Specifies the subresource from a 1D texture to use in a render-target view.
D3D11_TEX1D_SRV	Specifies the subresource from a 1D texture to use in a shader-resource view.
D3D11_TEX1D_UAV	Describes a unordered-access 1D texture resource.
D3D11_TEX2D_ARRAY_DSV	Specifies the subresources from an array 2D textures that are accessible to a depth-stencil view.
D3D11_TEX2D_ARRAY_RTV	Specifies the subresources from an array of 2D textures to use in a render-target view.
D3D11_TEX2D_ARRAY_RTV1	Describes the subresources from an array of 2D textures to use in a render-target view.
D3D11_TEX2D_ARRAY_SRV	Specifies the subresources from an array of 2D textures to use in a shader-resource view.
D3D11_TEX2D_ARRAY_SRV1	Describes the subresources from an array of 2D textures to use in a shader-resource view.
D3D11_TEX2D_ARRAY_UAV	Describes an array of unordered-access 2D texture resources.
D3D11_TEX2D_ARRAY_UAV1	Describes an array of unordered-access 2D texture resources.
D3D11_TEX2D_DSV	Specifies the subresource from a 2D texture that is accessible to a depth-stencil view.
D3D11_TEX2D_RTV	Specifies the subresource from a 2D texture to use in a render-target view.
D3D11_TEX2D_RTV1	Describes the subresource from a 2D texture to use in a render-target view.
D3D11_TEX2D_SRV	Specifies the subresource from a 2D texture to use in a shader-resource view.
D3D11_TEX2D_SRV1	Describes the subresource from a 2D texture to use in a shader-resource view.

TOPIC	DESCRIPTION
D3D11_TEX2D_UAV	Describes a unordered-access 2D texture resource.
D3D11_TEX2D_UAV1	Describes a unordered-access 2D texture resource.
D3D11_TEX2DMS_ARRAY_DSV	Specifies the subresources from an array of multisampled 2D textures for a depth-stencil view.
D3D11_TEX2DMS_ARRAY_RTV	Specifies the subresources from a an array of multisampled 2D textures to use in a render-target view.
D3D11_TEX2DMS_ARRAY_SRV	Specifies the subresources from an array of multisampled 2D textures to use in a shader-resource view.
D3D11_TEX2DMS_DSV	Specifies the subresource from a multisampled 2D texture that is accessible to a depth-stencil view.
D3D11_TEX2DMS_RTV	Specifies the subresource from a multisampled 2D texture to use in a render-target view.
D3D11_TEX2DMS_SRV	Specifies the subresources from a multisampled 2D texture to use in a shader-resource view.
D3D11_TEX3D_RTV	Specifies the subresources from a 3D texture to use in a render-target view.
D3D11_TEX3D_SRV	Specifies the subresources from a 3D texture to use in a shader-resource view.
D3D11_TEX3D_UAV	Describes a unordered-access 3D texture resource.
D3D11_TEXCUBE_ARRAY_SRV	Specifies the subresources from an array of cube textures to use in a shader-resource view.
D3D11_TEXCUBE_SRV	Specifies the subresource from a cube texture to use in a shader-resource view.
D3D11_TEXTURE1D_DESC	Describes a 1D texture.
D3D11_TEXTURE2D_DESC	Describes a 2D texture.
D3D11_TEXTURE2D_DESC1	Describes a 2D texture.
D3D11_TEXTURE3D_DESC	Describes a 3D texture.
D3D11_TEXTURE3D_DESC1	Describes a 3D texture.
D3D11_TILE_REGION_SIZE	Describes the size of a tiled region.
D3D11_TILED_RESOURCE_COORDINATE	Describes the coordinates of a tiled resource.
D3D11_TILE_SHAPE	Describes the shape of a tile by specifying its dimensions.

TOPIC	DESCRIPTION
D3D11_UNORDERED_ACCESS_VIEW_DESC	Specifies the subresources from a resource that are accessible using an unordered-access view.
D3D11_UNORDERED_ACCESS_VIEW_DESC1	Describes the subresources from a resource that are accessible using an unordered-access view.

Related topics

[Resource Reference](#)

Resource Enumerations (Direct3D 11 Graphics)

2/4/2021 • 2 minutes to read • [Edit Online](#)

Enumerations are used to specify information about how resources are created and accessed during rendering.

In this section

TOPIC	DESCRIPTION
D3D11_BIND_FLAG	Identifies how to bind a resource to the pipeline.
D3D11_BUFFEREX_SRV_FLAG	Identifies how to view a buffer resource.
D3D11_BUFFER_UAV_FLAG	Identifies unordered-access view options for a buffer resource.
D3D11_CHECK_MULTISAMPLE_QUALITY_LEVELS_FLAG	Identifies how to check multisample quality levels.
D3D11_CPU_ACCESS_FLAG	Specifies the types of CPU access allowed for a resource.
D3D11_DSV_DIMENSION	Specifies how to access a resource used in a depth-stencil view.
D3D11_DSV_FLAG	Depth-stencil view options.
D3D11_MAP	Identifies a resource to be accessed for reading and writing by the CPU. Applications may combine one or more of these flags.
D3D11_MAP_FLAG	Specifies how the CPU should respond when an application calls the ID3D11DeviceContext::Map method on a resource that is being used by the GPU.
D3D11_RESOURCE_DIMENSION	Identifies the type of resource being used.
D3D11_RESOURCE_MISC_FLAG	Identifies options for resources.
D3D11_RTV_DIMENSION	These flags identify the type of resource that will be viewed as a render target.
D3D11_SRV_DIMENSION	These flags identify the type of resource that will be viewed as a shader resource.
D3D11_STANDARD_MULTISAMPLE_QUALITY_LEVELS	Specifies a multi-sample pattern type.
D3D11_TEXTURE_LAYOUT	Specifies texture layout options.
D3D11_TILE_COPY_FLAG	Identifies how to copy a tile.

TOPIC	DESCRIPTION
D3D11_TILE_MAPPING_FLAG	Identifies how to perform a tile-mapping operation.
D3D11_TILE_RANGE_FLAG	Specifies a range of tile mappings to use with ID3D11DeviceContext2::UpdateTiles .
D3D11_UAV_DIMENSION	Unordered-access view options.
D3D11_USAGE	Identifies expected resource use during rendering. The usage directly reflects whether a resource is accessible by the CPU and/or the graphics processing unit (GPU).

Related topics

[Resource Reference](#)

Shader Reference (Direct3D 11 Graphics)

2/4/2021 • 2 minutes to read • [Edit Online](#)

The Direct3D API defines several API elements to help you create and manage programmable shaders. Shaders are executable programs that are programmed exclusively using HLSL.

In this section

TOPIC	DESCRIPTION
Shader Interfaces	This section contains information about the shader interfaces.
Shader Functions	This section contains information about the shader functions.
Shader Structures	Structures are used to create and use shaders.
Shader Enumerations	Enumerations are used to specify information about shaders.

Related topics

[Direct3D 11 Reference](#)

Shader Interfaces (Direct3D 11 Graphics)

2/4/2021 • 3 minutes to read • [Edit Online](#)

This section contains information about the shader interfaces.

Each of these shader interfaces manages a compiled shader. The interface is created when a shader is compiled, and is then passed to various APIs that need access to a compiled shader; such as when binding a shader to a pipeline stage or getting a shader signature.

In this section

TOPIC	DESCRIPTION
ID3D11ClassInstance	This interface encapsulates an HLSL class.
ID3D11ClassLinkage	This interface encapsulates an HLSL dynamic linkage.
ID3D11ComputeShader	A compute-shader interface manages an executable program (a compute shader) that controls the compute-shader stage.
ID3D11DomainShader	A domain-shader interface manages an executable program (a domain shader) that controls the domain-shader stage.
ID3D11FunctionLinkingGraph	A function-linking-graph interface is used for constructing shaders that consist of a sequence of precompiled function calls that pass values to each other. <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"><p>[!Note]</p><p>This interface is part of the HLSL shader linking technology that you can use on all Direct3D 11 platforms to create precompiled HLSL functions, package them into libraries, and link them into full shaders at run time.</p></div>
ID3D11FunctionReflection	A function-reflection interface accesses function info. <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"><p>[!Note]</p><p>This interface is part of the HLSL shader linking technology that you can use on all Direct3D 11 platforms to create precompiled HLSL functions, package them into libraries, and link them into full shaders at run time.</p></div>

TOPIC	DESCRIPTION
ID3D11FunctionParameterReflection	<p>A function-parameter-reflection interface accesses function-parameter info.</p> <div data-bbox="826 265 1414 458" style="border: 1px solid black; padding: 5px;"> <p>[!Note]</p> <p>This interface is part of the HLSL shader linking technology that you can use on all Direct3D 11 platforms to create precompiled HLSL functions, package them into libraries, and link them into full shaders at run time.</p> </div>
ID3D11GeometryShader	<p>A geometry-shader interface manages an executable program (a geometry shader) that controls the geometry-shader stage.</p>
ID3D11HullShader	<p>A hull-shader interface manages an executable program (a hull shader) that controls the hull-shader stage.</p>
ID3D11LibraryReflection	<p>A library-reflection interface accesses library info.</p> <div data-bbox="826 893 1414 1087" style="border: 1px solid black; padding: 5px;"> <p>[!Note]</p> <p>This interface is part of the HLSL shader linking technology that you can use on all Direct3D 11 platforms to create precompiled HLSL functions, package them into libraries, and link them into full shaders at run time.</p> </div>
ID3D11Linker	<p>A linker interface is used to link a shader module.</p> <div data-bbox="826 1268 1414 1462" style="border: 1px solid black; padding: 5px;"> <p>[!Note]</p> <p>This interface is part of the HLSL shader linking technology that you can use on all Direct3D 11 platforms to create precompiled HLSL functions, package them into libraries, and link them into full shaders at run time.</p> </div>
ID3D11LinkingNode	<p>A linking-node interface is used for shader linking.</p> <div data-bbox="826 1641 1414 1834" style="border: 1px solid black; padding: 5px;"> <p>[!Note]</p> <p>This interface is part of the HLSL shader linking technology that you can use on all Direct3D 11 platforms to create precompiled HLSL functions, package them into libraries, and link them into full shaders at run time.</p> </div>

TOPIC	DESCRIPTION
ID3D11Module	<p>A module interface creates an instance of a module that is used for resource rebinding.</p> <div data-bbox="826 260 1414 451" style="border: 1px solid black; padding: 5px;"> <p>[!Note]</p> <p>This interface is part of the HLSL shader linking technology that you can use on all Direct3D 11 platforms to create precompiled HLSL functions, package them into libraries, and link them into full shaders at run time.</p> </div>
ID3D11ModuleInstance	<p>A module-instance interface is used for resource rebinding.</p> <div data-bbox="826 631 1414 822" style="border: 1px solid black; padding: 5px;"> <p>[!Note]</p> <p>This interface is part of the HLSL shader linking technology that you can use on all Direct3D 11 platforms to create precompiled HLSL functions, package them into libraries, and link them into full shaders at run time.</p> </div>
ID3D11PixelShader	<p>A pixel-shader interface manages an executable program (a pixel shader) that controls the pixel-shader stage.</p>
ID3D11ShaderReflection	<p>A shader-reflection interface accesses shader information.</p>
ID3D11ShaderReflectionConstantBuffer	<p>This shader-reflection interface provides access to a constant buffer.</p>
ID3D11ShaderReflectionType	<p>This shader-reflection interface provides access to variable type.</p>
ID3D11ShaderReflectionVariable	<p>This shader-reflection interface provides access to a variable.</p>
ID3D11ShaderTrace	<p>An ID3D11ShaderTrace interface implements methods for obtaining traces of shader executions.</p>
ID3D11ShaderTraceFactory	<p>An ID3D11ShaderTraceFactory interface implements a method for generating shader trace information objects.</p>
ID3D11VertexShader	<p>A vertex-shader interface manages an executable program (a vertex shader) that controls the vertex-shader stage.</p>

Related topics

[Shader Reference](#)

Shader Functions (Direct3D 11 Graphics)

2/4/2021 • 2 minutes to read • [Edit Online](#)

This section contains information about the shader functions.

In this section

TOPIC	DESCRIPTION
D3DDisassemble11Trace	Disassembles a section of compiled Microsoft High Level Shader Language (HLSL) code that is specified by shader trace steps.

Related topics

[Shader Reference](#)

Shader Structures (Direct3D 11 Graphics)

2/4/2021 • 2 minutes to read • [Edit Online](#)

Structures are used to create and use shaders.

In this section

TOPIC	DESCRIPTION
D3D11_CLASS_INSTANCE_DESC	Describes an HLSL class instance.
D3D11_COMPUTE_SHADER_TRACE_DESC	Describes an instance of a compute shader to trace.
D3D11_DOMAIN_SHADER_TRACE_DESC	Describes an instance of a domain shader to trace.
D3D11_FUNCTION_DESC	Describes a function.
D3D11_GEOMETRY_SHADER_TRACE_DESC	Describes an instance of a geometry shader to trace.
D3D11_HULL_SHADER_TRACE_DESC	Describes an instance of a hull shader to trace.
D3D11_LIBRARY_DESC	Describes a library.
D3D11_PARAMETER_DESC	Describes a function parameter.
D3D11_PIXEL_SHADER_TRACE_DESC	Describes an instance of a pixel shader to trace.
D3D11_SHADER_BUFFER_DESC	Describes a shader constant-buffer.
D3D11_SHADER_DESC	Describes a shader.
D3D11_SHADER_INPUT_BIND_DESC	Describes how a shader resource is bound to a shader input.
D3D11_SHADER_TRACE_DESC	Describes a shader-trace object.
D3D11_SHADER_TYPE_DESC	Describes a shader-variable type.
D3D11_SHADER_VARIABLE_DESC	Describes a shader variable.
D3D11_SIGNATURE_PARAMETER_DESC	Describes a shader signature.
D3D11_TRACE_REGISTER	Describes a trace register.
D3D11_TRACE_STATS	Specifies statistics about a trace.
D3D11_TRACE_STEP	Describes a trace step, which is an instruction.
D3D11_TRACE_VALUE	Describes a trace value.

TOPIC	DESCRIPTION
D3D11_VERTEX_SHADER_TRACE_DESC	Describes an instance of a vertex shader to trace.

Related topics

[Shader Reference](#)

Shader Enumerations (Direct3D 11 Graphics)

2/4/2021 • 2 minutes to read • [Edit Online](#)

Enumerations are used to specify information about shaders.

In this section

TOPIC	DESCRIPTION
D3D11_CBUFFER_TYPE	Indicates a constant buffer's type.
D3D_PARAMETER_FLAGS	Indicates semantic flags for function parameters.
D3D11_RESOURCE_RETURN_TYPE	Indicates return value type.
D3D11_SHADER_TYPE	Identifies a shader type for tracing.
D3D11_SHADER_VERSION_TYPE	Indicates shader type.
D3D11_TESSELLATOR_DOMAIN	Domain options for tessellator data.
D3D11_TESSELLATOR_PARTITIONING	Partitioning options.
D3D11_TESSELLATOR_OUTPUT_PRIMITIVE	Output primitive types.
D3D11_TRACE_GS_INPUT_PRIMITIVE	Identifies the type of geometry shader input primitive.
D3D11_TRACE_REGISTER_TYPE	Identifies a type of trace register.

Related topics

[Shader Reference](#)

10Level9 Reference

2/4/2021 • 2 minutes to read • [Edit Online](#)

This section specifies the differences between each 10Level9 feature level and the D3D_FEATURE_LEVEL_11_0 and higher feature level for the [ID3D11Device](#) and [ID3D11DeviceContext](#) methods.

In this section

TOPIC	DESCRIPTION
10Level9 ID3D11Device Methods	This section lists the differences between each 10Level9 feature level and the D3D_FEATURE_LEVEL_11_0 and higher feature level for the ID3D11Device methods.
10Level9 ID3D11DeviceContext Methods	This section lists the differences between each 10Level9 feature level and the D3D_FEATURE_LEVEL_11_0 and higher feature level for the ID3D11DeviceContext methods.

Related topics

[Direct3D 11 Reference](#)

10Level9 ID3D11Device Methods

11/2/2020 • 4 minutes to read • [Edit Online](#)

This section lists the differences between each 10Level9 feature level and the D3D_FEATURE_LEVEL_11_0 and higher feature level for the [ID3D11Device](#) methods.

- [ID3D11Device::CheckCounter](#)
- [ID3D11Device::CheckFormatSupport](#)
- [ID3D11Device::CheckMultisampleQualityLevels](#)
- [ID3D11Device::CreateBlendState](#)
- [ID3D11Device::CreateBlendState1](#)
- [ID3D11Device::CreateBuffer](#)
- [ID3D11Device::CreateCounter](#)
- [ID3D11Device::CreateDepthStencilView](#)
- [ID3D11Device::CreateDomainShader](#)
- [ID3D11Device::CreateGeometryShader](#)
- [ID3D11Device::CreateGeometryShaderWithStreamOutput](#)
- [ID3D11Device::CreateHullShader](#)
- [ID3D11Device::CreateInputLayout](#)
- [ID3D11Device::CreatePixelShader](#)
- [ID3D11Device::CreatePredicate](#)
- [ID3D11Device::CreateQuery](#)
- [ID3D11Device::CreateRasterizerState](#)
- [ID3D11Device::CreateRenderTargetView](#)
- [ID3D11Device::CreateSamplerState](#)
- [ID3D11Device::CreateShaderResourceView](#)
- [ID3D11Device::CreateTexture1D](#)
- [ID3D11Device::CreateTexture2D](#)
- [ID3D11Device::CreateTexture3D](#)
- [ID3D11Device::CreateUnorderedAccessView](#)
- [ID3D11Device::CreateVertexShader](#)
- [ID3D11Device::OpenSharedResource](#)
- [Related topics](#)

ID3D11Device::CheckCounter

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Device-dependent counters are optionally supported. Use ID3D11Device::CheckCounterInfo to determine support.\$(REMOVE)\$
D3D_FEATURE_LEVEL_9_2	Device-dependent counters are optionally supported. Use ID3D11Device::CheckCounterInfo to determine support.\$(REMOVE)\$
D3D_FEATURE_LEVEL_9_3	Device-dependent counters are optionally supported. Use ID3D11Device::CheckCounterInfo to determine support.\$(REMOVE)\$

ID3D11Device::CheckFormatSupport

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	See format support by feature level \${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11Device::CheckMultisampleQualityLevels

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Feature levels make no guarantees concerning MSAA support. \${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11Device::CreateBlendState

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	AlphaToCoverageEnable must be FALSE . The first four BlendEnables must all have the same value. D3D11_BLEND_SRC_ALPHASAT not supported. Dual-source color blend not supported (any SrcBlend or DestBlend with SRC1 in the name)
D3D_FEATURE_LEVEL_9_2	AlphaToCoverageEnable must be FALSE . The first four BlendEnables must all have the same value. The first four RenderTargetWriteMasks must all have the same value. D3D11_BLEND_SRC_ALPHASAT not supported. Dual-source color blend not supported (any SrcBlend or DestBlend with SRC1 in the name)
D3D_FEATURE_LEVEL_9_3	AlphaToCoverageEnable must be FALSE . The first four BlendEnables must all have the same value. D3D11_BLEND_SRC_ALPHASAT not supported. Dual-source color blend not supported (any SrcBlend or DestBlend with SRC1 in the name)
D3D_FEATURE_LEVEL_10_0	Adds alpha-to-coverage

ID3D11Device::CreateBlendState1

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Unsupported
D3D_FEATURE_LEVEL_9_2	Unsupported
D3D_FEATURE_LEVEL_9_3	Unsupported

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_10_0	The <i>OutputMergerLogicOp</i> member has been added to D3D11_FEATURE_DATA_D3D11_OPTIONS , to determine support for logical operations (bitwise logic operations between pixel shader output and render target contents, refer to D3D11_RENDER_TARGET_BLEND_DESC1).

ID3D11Device::CreateBuffer

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Buffers can't have render target views. Buffers must have exactly one of D3D11_BIND_VERTEX_BUFFER, D3D11_BIND_INDEX_BUFFER, or D3D11_BIND_CONSTANT_BUFFER. Only allows index buffers with the DXGI_FORMAT_R16_UINT format.
D3D_FEATURE_LEVEL_9_2	Buffers can't have render target views. Buffers must have exactly one of D3D11_BIND_VERTEX_BUFFER, D3D11_BIND_INDEX_BUFFER, or D3D11_BIND_CONSTANT_BUFFER.
D3D_FEATURE_LEVEL_9_3	Allows index buffers with the DXGI_FORMAT_R16_UINT and DXGI_FORMAT_R32_UINT formats like D3D_FEATURE_LEVEL_10_0 and higher. \${REMOVE}\$

ID3D11Device::CreateCounter

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* feature level.\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11Device::CreateDepthStencilView

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Does not support two-sided stencil.\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11Device::CreateDomainShader

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* or 10.* feature level. \${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	
D3D_FEATURE_LEVEL_10_0	
D3D_FEATURE_LEVEL_10_1	

ID3D11Device::CreateGeometryShader

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* feature level. \${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11Device::CreateGeometryShaderWithStreamOutput

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* feature level. \${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11Device::CreateHullShader

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* or 10.* feature level. \${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	
D3D_FEATURE_LEVEL_10_0	
D3D_FEATURE_LEVEL_10_1	

ID3D11Device::CreateInputLayout

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Does not support D3D11_INPUT_PER_INSTANCE_DATA
D3D_FEATURE_LEVEL_9_2	Does not support D3D11_INPUT_PER_INSTANCE_DATA
D3D_FEATURE_LEVEL_9_3	Vertex stream zero must have D3D11_INPUT_PER_VERTEX_DATA, if any streams have D3D11_INPUT_PER_VERTEX_DATA

See the format support by [feature level chart](#) for details on what formats can be used for vertex data at each feature level.

ID3D11Device::CreatePixelShader

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Must use ps_4_0_level_9_1
D3D_FEATURE_LEVEL_9_2	Must use ps_4_0_level_9_1
D3D_FEATURE_LEVEL_9_3	Must use ps_4_0_level_9_3 or ps_4_0_level_9_1

ID3D11Device::CreatePredicate

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* feature level. {REMOVE}
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11Device::CreateQuery

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Event queries are supported. Timestamp queries are optional: call CreateQuery to determine support.
D3D_FEATURE_LEVEL_9_2	Event and occlusion queries are supported. Timestamp queries are optional: call CreateQuery to determine support.
D3D_FEATURE_LEVEL_9_3	Event and occlusion queries are supported. Timestamp queries are optional: call CreateQuery to determine support.

ID3D11Device::CreateRasterizerState

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	DepthClipEnable must be TRUE . DepthBiasClamp must be set to 0.\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11Device::CreateRenderTargetView

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Can only support render target views of Texture2D objects.\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11Device::CreateSamplerState

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Comparison filter is not supported. Border color must be within [0,1] Min LOD cannot be fractional Max LOD must be FLT_MAX Maximum anisotropy is 2. D3D11_TEXTURE_ADDRESS_MIRRORONCE not supported.
D3D_FEATURE_LEVEL_9_2	Comparison filter is not supported. Border color must be within [0,1] Min LOD cannot be fractional Max LOD must be FLT_MAX Maximum anisotropy is 16. \${REMOVE}\$
D3D_FEATURE_LEVEL_9_3	

ID3D11Device::CreateShaderResourceView

FEATURE LEVEL	MOSTDETAILEDMIP PLUS MIPLEVELS MUST INCLUDE LOWEST LOD (SMALLEST SUBRESOURCE)	VIEW MUST INCLUDE ALL RESOURCE ARRAY ELEMENTS
D3D_FEATURE_LEVEL_9_1	Yes	yes
D3D_FEATURE_LEVEL_9_2	Yes	Yes
D3D_FEATURE_LEVEL_9_3	Yes	Yes

ID3D11Device::CreateTexture1D

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* feature level. \${REMOVE}
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11Device::CreateTexture2D

Texture2D resources have limits on their width and height that differ across [feature levels](#). In feature levels 9_3, the following are guaranteed minima, and individual implementations may exceed the requirements.

FEATURE LEVEL	IF MIPCOUNT > 1, DIMENSIONS MUST BE INTEGRAL POWER OF TWO	MINIMUM SUPPORTED TEXTURE DIMENSION	CUBE TEXTURES DIMENSIONS MUST BE POWER OF TWO	IF MISC_TEXTURECUBE IS SET, THE ARRAYSIZE IS:	IF MISC_TEXTURECUBE IS NOT SET, THE ARRAYSIZE IS.
D3D_FEATURE_LEVEL_9_1	Yes	2048	Yes	6	1
D3D_FEATURE_LEVEL_9_2	Yes	2048	Yes	6	1
D3D_FEATURE_LEVEL_9_3	Yes	4096	Yes	6	1

In the previous table, the full name of MISC_TEXTURECUBE is [D3D11_RESOURCE_MISC_TEXTURECUBE](#).

The following are true for all 9_* [feature levels](#):

- When using D3D11_USAGE_DEFAULT or D3D11_USAGE_IMMUTABLE, BindFlags cannot be zero.
- When using D3D11_BIND_DEPTH_STENCIL, MipLevels must be 1.
- When using D3D11_BIND_SHADER_RESOURCE, SampleDesc.Count must be 1.
- When using D3D11_BIND_PRESENT, resource cannot have D3D11_BIND_SHADER_RESOURCE.
- When using D3D10_DDI_RESOURCE_MISC_SHARED, Format cannot be DXGI_FORMAT_R8G8B8A8_UNORM or DXGI_FORMAT_R8G8B8A8_UNORM_SRGB.

ID3D11Device::CreateTexture3D

FEATURE LEVEL	MAXIMUM DIMENSION (ANY AXIS)	DIMENSIONS MUST BE POWER OF TWO
D3D_FEATURE_LEVEL_9_1	256	Yes
D3D_FEATURE_LEVEL_9_2	512	Yes
D3D_FEATURE_LEVEL_9_3	512	Yes

If resource is D3D11_USAGE_DEFAULT or D3D11_USAGE_IMMUTABLE, BindFlags cannot be zero.

ID3D11Device::CreateUnorderedAccessView

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* feature level.\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11Device::CreateVertexShader

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Must use vs_4_0_level_9_1
D3D_FEATURE_LEVEL_9_2	Must use vs_4_0_level_9_1
D3D_FEATURE_LEVEL_9_3	Must use vs_4_0_level_9_3 or vs_4_0_level_9_1

ID3D11Device::OpenSharedResource

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Use ID3D11Device::CheckFeatureSupport with the D3D11_FEATURE_FORMAT_SUPPORT2 value and the D3D11_FEATURE_DATA_FORMAT_SUPPORT2 structure to determine if a format can be shared. If the format can be shared, CheckFeatureSupport returns the D3D11_FORMAT_SUPPORT2_SHAREABLE flag.
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	<p>[!Note]</p> <p>[DXGI_FORMAT_R8G8B8A8_UNORM] (/windows/desktop/api/dxgiformat/ne-dxgiformat-dxgi_format) and DXGI_FORMAT_R8G8B8A8_UNORM_SRGB are never shareable when using feature level 9, even if the device indicates optional feature support for D3D11_FORMAT_SUPPORT_SHAREABLE. Attempting to create shared resources with DXGI formats DXGI_FORMAT_R8G8B8A8_UNORM and DXGI_FORMAT_R8G8B8A8_UNORM_SRGB will always fail unless the feature level is 10_0 or higher.</p>

Related topics

[10Level9 Reference](#)

10Level9 ID3D11DeviceContext Methods

2/22/2020 • 6 minutes to read • [Edit Online](#)

This section lists the differences between each 10Level9 feature level and the D3D_FEATURE_LEVEL_11_0 and higher feature level for the [ID3D11DeviceContext](#) methods.

- [ID3D11DeviceContext::CopySubresourceRegion](#)
- [ID3D11DeviceContext::CopyResource](#)
- [ID3D11DeviceContext::CopyStructureCount](#)
- [ID3D11DeviceContext::ClearUnorderedAccessViewFloat](#)
- [ID3D11DeviceContext::ClearUnorderedAccessViewUint](#)
- [ID3D11DeviceContext::ClearRenderTargetView](#)
- [ID3D11DeviceContext::CSSetConstantBuffers](#)
- [ID3D11DeviceContext::CSSetSamplers](#)
- [ID3D11DeviceContext::CSSetShader](#)
- [ID3D11DeviceContext::CSSetShaderResources](#)
- [ID3D11DeviceContext::CSSetUnorderedAccessViews](#)
- [ID3D11DeviceContext::Dispatch](#)
- [ID3D11DeviceContext::DispatchIndirect](#)
- [ID3D11DeviceContext::Draw](#)
- [ID3D11DeviceContext::DrawAuto](#)
- [ID3D11DeviceContext::DrawIndexed](#)
- [ID3D11DeviceContext::DrawIndexedInstanced](#)
- [ID3D11DeviceContext::DrawIndexedInstancedIndirect](#)
- [ID3D11DeviceContext::DrawInstanced](#)
- [ID3D11DeviceContext::DrawInstancedIndirect](#)
- [ID3D11DeviceContext::DSSetConstantBuffers](#)
- [ID3D11DeviceContext::DSSetSamplers](#)
- [ID3D11DeviceContext::DSSetShader](#)
- [ID3D11DeviceContext::DSSetShaderResources](#)
- [ID3D11DeviceContext::GSSetConstantBuffers](#)
- [ID3D11DeviceContext::GSSetSamplers](#)
- [ID3D11DeviceContext::GSSetShader](#)
- [ID3D11DeviceContext::GSSetShaderResources](#)
- [ID3D11DeviceContext::HSSetConstantBuffers](#)
- [ID3D11DeviceContext::HSSetSamplers](#)
- [ID3D11DeviceContext::HSSetShader](#)
- [ID3D11DeviceContext::HSSetShaderResources](#)
- [ID3D11DeviceContext::IASetIndexBuffer](#)
- [ID3D11DeviceContext::IASetPrimitiveTopology](#)
- [ID3D11DeviceContext::OMSetBlendState](#)
- [ID3D11DeviceContext::OMSetRenderTargets](#)
- [ID3D11DeviceContext::OMSetRenderTargetsAndUnorderedAccessViews](#)
- [ID3D11DeviceContext::PSSetConstantBuffers](#)

- [ID3D11DeviceContext::PSSetSamplers](#)
- [ID3D11DeviceContext::PSSetShader](#)
- [ID3D11DeviceContext::PSSetShaderResources](#)
- [ID3D11DeviceContext::RSSetScissorRects](#)
- [ID3D11DeviceContext::RSSetViewports](#)
- [ID3D11DeviceContext::SetPredication](#)
- [ID3D11DeviceContext::SOSetTargets](#)
- [ID3D11DeviceContext::VSSetConstantBuffers](#)
- [ID3D11DeviceContext::VSSetSamplers](#)
- [ID3D11DeviceContext::VSSetShader](#)
- [ID3D11DeviceContext::VSSetShaderResources](#)
- [Related topics](#)

ID3D11DeviceContext::CopySubresourceRegion

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Only Texture2D and buffers may be copied within GPU-accessible memory.
D3D_FEATURE_LEVEL_9_2	Texture3D cannot be copied from GPU-accessible memory to CPU-accessible memory. Any resource that has only D3D10_BIND_SHADER_RESOURCE cannot be copied from GPU-accessible memory to CPU-accessible memory. You can't copy mipmapped volume textures. \${REMOVE}\$
D3D_FEATURE_LEVEL_9_3	

ID3D11DeviceContext::CopyResource

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Only Texture2D and buffers may be copied within GPU-accessible memory.
D3D_FEATURE_LEVEL_9_2	Texture3D cannot be copied from GPU-accessible memory to CPU-accessible memory. Any resource that has only D3D10_BIND_SHADER_RESOURCE cannot be copied from GPU-accessible memory to CPU-accessible memory. \${REMOVE}\$
D3D_FEATURE_LEVEL_9_3	

ID3D11DeviceContext::CopyStructureCount

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* feature level.\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11DeviceContext::ClearUnorderedAccessViewFloat

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* feature level.\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11DeviceContext::ClearUnorderedAccessViewUint

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* feature level.\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11DeviceContext::ClearRenderTargetView

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Only the first array slice will be cleared. Applications should create a render target view for each face or array slice, then clear each view individually.\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11DeviceContext::CSSetConstantBuffers

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* feature level.\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11DeviceContext::CSSetSamplers

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* feature level.\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11DeviceContext::CSSetShader

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* feature level.\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11DeviceContext::CSSetShaderResources

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* feature level.\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11DeviceContext::CSSetUnorderedAccessViews

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* feature level.\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11DeviceContext::Dispatch

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* feature level.\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11DeviceContext::DispatchIndirect

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* feature level.\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11DeviceContext::Draw

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Number of primitives may not exceed 65535. Textures cannot repeat over one primitive more than 128 times.
D3D_FEATURE_LEVEL_9_2	Number of primitives may not exceed 1048575. Textures cannot repeat over one primitive more than 2048 times.
D3D_FEATURE_LEVEL_9_3	Number of primitives may not exceed 1048575. Textures cannot repeat over one primitive more than 8192 times.

ID3D11DeviceContext::DrawAuto

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* feature level.\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11DeviceContext::DrawIndexed

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Number of primitives may not exceed 65535. Textures cannot repeat over one primitive more than 128 times. Index values cannot exceed 65534. Indexed point lists not supported.
D3D_FEATURE_LEVEL_9_2	Number of primitives may not exceed 1048575. Textures cannot repeat over one primitive more than 2048 times. Index values cannot exceed 1048575. Indexed point lists not supported.
D3D_FEATURE_LEVEL_9_3	Number of primitives may not exceed 1048575. Textures cannot repeat over one primitive more than 8192 times. Index values cannot exceed 1048575. Indexed point lists not supported.

ID3D11DeviceContext::DrawIndexedInstanced

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_3	<p>Number of primitives may not exceed 1048575. Textures cannot repeat over one primitive more than 8192 times. Index values cannot exceed 1048575.</p> <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <p>[!Note]</p> <p>When you call the DrawIndexedInstanced method with a vertex shader that is bound to the pipeline and that doesn't import any per-instance data, some Direct3D 9 graphics hardware might not draw anything. In particular, if the vertex shader does not use any per-instance data, calling DrawIndexedInstanced with 1 instance is not equivalent to calling Draw.</p> </div>

ID3D11DeviceContext::DrawIndexedInstancedIndirect

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* or 10.* feature level.\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	
D3D_FEATURE_LEVEL_10_0	
D3D_FEATURE_LEVEL_10_1	

ID3D11DeviceContext::DrawInstanced

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* feature level.\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11DeviceContext::DrawInstancedIndirect

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* or 10.* feature level.\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_10_0	
D3D_FEATURE_LEVEL_10_1	

ID3D11DeviceContext::DSSetConstantBuffers

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* or 10.* feature level.{REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	
D3D_FEATURE_LEVEL_10_0	
D3D_FEATURE_LEVEL_10_1	

ID3D11DeviceContext::DSSetSamplers

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* or 10.* feature level.{REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	
D3D_FEATURE_LEVEL_10_0	
D3D_FEATURE_LEVEL_10_1	

ID3D11DeviceContext::DSSetShader

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* or 10.* feature level.{REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	
D3D_FEATURE_LEVEL_10_0	
D3D_FEATURE_LEVEL_10_1	

ID3D11DeviceContext::DSSetShaderResources

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* or 10.* feature level.\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	
D3D_FEATURE_LEVEL_10_0	
D3D_FEATURE_LEVEL_10_1	

ID3D11DeviceContext::GSSetConstantBuffers

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* feature level.\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11DeviceContext::GSSetSamplers

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* feature level.\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11DeviceContext::GSSetShader

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* feature level.\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11DeviceContext::GSSetShaderResources

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* feature level.\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_3	

ID3D11DeviceContext::HSSetConstantBuffers

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* or 10.* feature level.{REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	
D3D_FEATURE_LEVEL_10_0	
D3D_FEATURE_LEVEL_10_1	

ID3D11DeviceContext::HSSetSamplers

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* or 10.* feature level.{REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	
D3D_FEATURE_LEVEL_10_0	
D3D_FEATURE_LEVEL_10_1	

ID3D11DeviceContext::HSSetShader

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* or 10.* feature level.{REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	
D3D_FEATURE_LEVEL_10_0	
D3D_FEATURE_LEVEL_10_1	

ID3D11DeviceContext::HSSetShaderResources

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* or 10.* feature level.\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	
D3D_FEATURE_LEVEL_10_0	
D3D_FEATURE_LEVEL_10_1	

ID3D11DeviceContext::IASetIndexBuffer

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Format is allowed to be different from that specified at buffer creation, but an expensive translation will be incurred. Only allows index buffers with the DXGI_FORMAT_R16_UINT format.
D3D_FEATURE_LEVEL_9_2	Format is allowed to be different from that specified at buffer creation, but an expensive translation will be incurred.
D3D_FEATURE_LEVEL_9_3	Allows index buffers with the DXGI_FORMAT_R16_UINT and DXGI_FORMAT_R32_UINT formats like D3D_FEATURE_LEVEL_10_0 and higher. \${REMOVE}\$

ID3D11DeviceContext::IASetPrimitiveTopology

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Primitive topologies with adjacency are not supported\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11DeviceContext::OMSetBlendState

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	SampleMask cannot be zero\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11DeviceContext::OMSetRenderTargets

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Only one render target supported\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	Only four render targets supported, and all bound resources must have same bit depth.

ID3D11DeviceContext::OMSetRenderTargetsAndUnorderedAccessViews

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* feature level.\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11DeviceContext::PSSetConstantBuffers

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	See feature level 10.0, but total number of constants used by shader cannot exceed 32\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11DeviceContext::PSSetSamplers

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	No more than 16 samplers can be bound\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11DeviceContext::PSSetShader

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Only ps_4_0_level_9_1\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	Only ps_4_0_level_9_3 or ps_4_0_level_9_1

ID3D11DeviceContext::PSSetShaderResources

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	No more than 8 simultaneously bound shader resources\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11DeviceContext::RSSetScissorRects

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Only the zeroth scissor rect is available\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11DeviceContext::RSSetViewports

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Only the zeroth viewport is available\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

Even though you specify float values to the members of the [D3D11_VIEWPORT](#) structure for the *pViewports* array in a call to [ID3D11DeviceContext::RSSetViewports](#) for feature levels 9_x, **RSSetViewports** uses DWORDs internally. Because of this behavior, when you use a negative top left corner for the viewport, the call to **RSSetViewports** for feature levels 9_x fails. This failure occurs because **RSSetViewports** for 9_x casts the floating point values into unsigned integers without validation, which results in integer overflow.

The call to [ID3D11DeviceContext::RSSetViewports](#) for feature levels 10_x and 11_x works as you expect even when you use a negative top left corner for the viewport.

ID3D11DeviceContext::SetPredication

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* feature level.\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11DeviceContext::SOSetTargets

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* feature level.\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11DeviceContext::VSSetConstantBuffers

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	See feature level 10.0, but total number of constants used by shader cannot exceed 255\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11DeviceContext::VSSetSamplers

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* feature level.\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

ID3D11DeviceContext::VSSetShader

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Only vs_4_0_level_9_1\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	Only vs_4_0_level_9_3 or vs_4_0_level_9_1

ID3D11DeviceContext::VSSetShaderResources

FEATURE LEVEL	BEHAVIOR DIFFERENCES
D3D_FEATURE_LEVEL_9_1	Not supported on any 9.* feature level.\${REMOVE}\$
D3D_FEATURE_LEVEL_9_2	
D3D_FEATURE_LEVEL_9_3	

Related topics

Direct3D 11 Return Codes

11/2/2020 • 2 minutes to read • [Edit Online](#)

Return codes from API functions.

HRESULT	DESCRIPTION
D3D11_ERROR_FILE_NOT_FOUND (0x887C0002)	The file was not found.
D3D11_ERROR_TOO_MANY_UNIQUE_STATE_OBJECTS (0x887C0001)	There are too many unique instances of a particular type of state object.
D3D11_ERROR_TOO_MANY_UNIQUE_VIEW_OBJECTS (0x887C0003)	There are too many unique instances of a particular type of view object.
D3D11_ERROR_DEFERRED_CONTEXT_MAP_WITHOUT_INITIAL_DISCARD (0x887C0004)	The first call to <a>ID3D11DeviceContext::Map after either <a>ID3D11Device::CreateDeferredContext or <a>ID3D11DeviceContext::FinishCommandList per Resource was not D3D11_MAP_WRITE_DISCARD.
D3DERR_INVALIDCALL (replaced with DXGI_ERROR_INVALID_CALL) (0x887A0001)	The method call is invalid. For example, a method's parameter may not be a valid pointer.
D3DERR_WASSTILLDRAWING (replaced with DXGI_ERROR_WAS_STILL_DRAWING) (0x887A000A)	The previous blit operation that is transferring information to or from this surface is incomplete.
E_FAIL (0x80004005)	Attempted to create a device with the debug layer enabled and the layer is not installed.
E_INVALIDARG (0x80070057)	An invalid parameter was passed to the returning function.
E_OUTOFMEMORY (0x8007000E)	Direct3D could not allocate sufficient memory to complete the call.
E_NOTIMPL (0x80004001)	The method call isn't implemented with the passed parameter combination.
S_FALSE ((HRESULT)1L)	Alternate success value, indicating a successful but nonstandard completion (the precise meaning depends on context).
S_OK ((HRESULT)0L)	No error occurred.

For more return codes, see DXGI_ERROR.

Related topics

- Direct3D 11 Reference

Common Version Reference

2/4/2021 • 2 minutes to read • [Edit Online](#)

The Direct3D API defines several API elements that are common to the Direct3D 12, Direct3D 11, Direct3D 10, and Direct3D 10.1. You can use these API elements in your code for any of these Direct3D versions. These API elements are known as version neutral.

In this section

TOPIC	DESCRIPTION
Common Version Interfaces	This section contains information about the common version interfaces.
Common Version Structures	This section contains information about the common version structures.
Common Version Enumerations	This section contains information about the common version enumerations.

Related topics

[Direct3D 11 Reference](#)

Common version interfaces

11/2/2020 • 2 minutes to read • [Edit Online](#)

This section contains information about the common version interfaces.

In this section

TOPIC	DESCRIPTION
ID3D10Blob	This interface is used to return arbitrary-length data.
ID3DBlob	This interface is used to return data of arbitrary length.
ID3DDestructionNotifier	Used to register for callbacks when a Direct 3D nano-COM object is destroyed.
ID3DInclude	ID3DInclude is an include interface that the user implements to allow an application to call user-overridable methods for opening and closing shader #include files.
ID3DUserDefinedAnnotation	The ID3DUserDefinedAnnotation interface enables an application to describe conceptual sections and markers within the application's code flow. An appropriately enabled tool, such as Microsoft Visual Studio Ultimate 2012, can display these sections and markers visually along the tool's Microsoft Direct3D time line, while the tool debugs the application. These visual notes allow users of such a tool to navigate to parts of the time line that are of interest, or to understand what set of Direct3D calls are produced by certain sections of the application's code.

Related topics

[Common version reference](#)

Common Version Structures

2/4/2021 • 2 minutes to read • [Edit Online](#)

This section contains information about the common version structures.

In this section

TOPIC	DESCRIPTION
D3D_SHADER_MACRO	Defines a shader macro.

Related topics

[Common Version Reference](#)

Common Version Enumerations

2/4/2021 • 2 minutes to read • [Edit Online](#)

This section contains information about the common version enumerations.

In this section

TOPIC	DESCRIPTION
D3D_CBUFFER_TYPE	Values that identify the intended use of constant-buffer data.
D3D_DRIVER_TYPE	Driver type options.
D3D_FEATURE_LEVEL	Describes the set of features targeted by a Direct3D device.
D3D_INCLUDE_TYPE	Values that indicate the location of a shader #include file.
D3D_INTERPOLATION_MODE	Specifies interpolation mode, which affects how values are calculated during rasterization.
D3D_MIN_PRECISION	Values that indicate the minimum desired interpolation precision.
D3D_NAME	Values that identify shader parameters that use system-value semantics.
D3D_PRIMITIVE	Values that indicate how the pipeline interprets geometry or hull shader input primitives.
D3D_PRIMITIVE_TOPOLOGY	Values that indicate how the pipeline interprets vertex data that is bound to the input-assembler stage . These primitive topology values determine how the vertex data is rendered on screen.
D3D_REGISTER_COMPONENT_TYPE	Values that identify the data types that can be stored in a register.
D3D_RESOURCE_RETURN_TYPE	Values that identify the return type of a resource.
D3D_SHADER_CBUFFER_FLAGS	Values that identify the intended use of a constant-data buffer.
D3D_SHADER_INPUT_FLAGS	Values that identify shader-input options.
D3D_SHADER_INPUT_TYPE	Values that identify resource types that can be bound to a shader and that are reflected as part of the resource description for the shader.
D3D_SHADER_VARIABLE_CLASS	Values that identify the class of a shader variable.
D3D_SHADER_VARIABLE_FLAGS	Values that identify information about a shader variable.

TOPIC	DESCRIPTION
D3D_SHADER_VARIABLE_TYPE	Values that identify various data, texture, and buffer types that can be assigned to a shader variable.
D3D_SRV_DIMENSION	Values that identify the type of resource to be viewed as a shader resource.
D3D_TESSELLATOR_DOMAIN	Values that identify domain options for tessellator data.
D3D_TESSELLATOR_OUTPUT_PRIMITIVE	Values that identify output primitive types.
D3D_TESSELLATOR_PARTITIONING	Values that identify partitioning options.

Related topics

[Common Version Reference](#)

CD3D11 Helper Structures

2/4/2021 • 2 minutes to read • [Edit Online](#)

Direct3D 11 defines several helper structures that you can use to create Direct3D structures. These helper structures behave like C++ classes.

In this section

TOPIC	DESCRIPTION
CD3D11_RECT	Represents a rectangle and provides convenience methods for creating rectangles.
CD3D11_BOX	Represents a box and provides convenience methods for creating boxes.
CD3D11_DEPTH_STENCIL_DESC	Represents a depth-stencil-state structure and provides convenience methods for creating depth-stencil-state structures.
CD3D11_BLEND_DESC	Represents a blend-state structure and provides convenience methods for creating blend-state structures.
CD3D11_RASTERIZER_DESC	Represents a rasterizer-state structure and provides convenience methods for creating rasterizer-state structures.
CD3D11_BUFFER_DESC	Represents a buffer and provides convenience methods for creating buffers.
CD3D11_TEXTURE1D_DESC	Represents a 1D texture and provides convenience methods for creating 1D textures.
CD3D11_TEXTURE2D_DESC	Represents a 2D texture and provides convenience methods for creating 2D textures.
CD3D11_TEXTURE3D_DESC	Represents a 3D texture and provides convenience methods for creating 3D textures.
CD3D11_SHADER_RESOURCE_VIEW_DESC	Represents a shader-resource view and provides convenience methods for creating shader-resource views.
CD3D11_RENDER_TARGET_VIEW_DESC	Represents a render-target view and provides convenience methods for creating render-target views.
CD3D11_VIEWPORT	Represents a viewport and provides convenience methods for creating viewports.
CD3D11_DEPTH_STENCIL_VIEW_DESC	Represents a depth-stencil view and provides convenience methods for creating depth-stencil views.

TOPIC	DESCRIPTION
CD3D11_UNORDERED_ACCESS_VIEW_DESC	Represents a unordered-access view and provides convenience methods for creating unordered-access views.
CD3D11_SAMPLER_DESC	Represents a sampler state and provides convenience methods for creating sampler states.
CD3D11_QUERY_DESC	Represents a query and provides convenience methods for creating queries.
CD3D11_COUNTER_DESC	Represents a counter and provides convenience methods for creating counters.

Related topics

[Direct3D 11 Reference](#)

D3DCSX 11 Reference

2/4/2021 • 2 minutes to read • [Edit Online](#)

This section contains reference information about the D3DCSX utility library, which you can use with a compute shader.

In this section

TOPIC	DESCRIPTION
D3DCSX 11 Interfaces	This section contains reference information about the COM interfaces provided by the D3DCSX utility library.
D3DCSX 11 Functions	This section contains information about the D3DCSX 11 functions.
D3DCSX 11 Structures	This section contains information about the D3DCSX 11 structures.
D3DCSX 11 Enumerations	This section contains information about D3DCSX 11 enumerations.

Related topics

[Direct3D 11 Reference](#)

[Direct3D 11 Reference](#)

D3DCSX 11 Interfaces

2/4/2021 • 2 minutes to read • [Edit Online](#)

This section contains reference information about the COM interfaces provided by the D3DCSX utility library.

In this section

TOPIC	DESCRIPTION
ID3DX11FFT	Encapsulates forward and inverse FFTs.
ID3DX11Scan	Scan context.
ID3DX11SegmentedScan	Segmented scan context.

Related topics

[D3DCSX 11 Reference](#)

D3DCSX 11 Functions

2/4/2021 • 2 minutes to read • [Edit Online](#)

This section contains information about the D3DCSX 11 functions.

In this section

TOPIC	DESCRIPTION
D3DX11CreateScan	Creates a scan context.
D3DX11CreateSegmentedScan	Creates a segmented scan context.
D3DX11CreateFFT	Creates an ID3DX11FFT COM interface object.
D3DX11CreateFFT1DComplex	Creates an ID3DX11FFT COM interface object.
D3DX11CreateFFT1DReal	Creates an ID3DX11FFT COM interface object.
D3DX11CreateFFT2DComplex	Creates an ID3DX11FFT COM interface object.
D3DX11CreateFFT2DReal	Creates an ID3DX11FFT COM interface object.
D3DX11CreateFFT3DComplex	Creates an ID3DX11FFT COM interface object.
D3DX11CreateFFT3DReal	Creates an ID3DX11FFT COM interface object.

Related topics

[D3DCSX 11 Reference](#)

D3DCSX 11 Structures

2/4/2021 • 2 minutes to read • [Edit Online](#)

This section contains information about the D3DCSX 11 structures.

In this section

TOPIC	DESCRIPTION
D3DX11_FFT_BUFFER_INFO	Describes buffer requirements for an FFT.
D3DX11_FFT_DESC	Describes an FFT.

Related topics

[D3DCSX 11 Reference](#)

D3DCSX 11 Enumerations

2/4/2021 • 2 minutes to read • [Edit Online](#)

This section contains information about D3DCSX 11 enumerations.

In this section

TOPIC	DESCRIPTION
D3DX11_FFT_CREATE_FLAG	FFT creation flags.
D3DX11_FFT_DATA_TYPE	FFT data types.
D3DX11_FFT_DIM_MASK	Number of dimensions for FFT data.
D3DX11_SCAN_DATA_TYPE	Type for scan data.
D3DX11_SCAN_DIRECTION	Direction to perform scan in.
D3DX11_SCAN_OPCODE	Scan opcodes.

Related topics

[D3DCSX 11 Reference](#)

D3DX 11 Reference

2/4/2021 • 2 minutes to read • [Edit Online](#)

The D3DX 11 API is described in this section.

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

In this section

TOPIC	DESCRIPTION
D3DX Interfaces	This section contains reference information for the component object model (COM) interfaces provided by the D3DX utility library.
D3DX Functions	This section contains information about the D3DX 11 functions.
D3DX Structures	This section contains information about the D3DX 11 structures.
D3DX Enumerations	This section contains information about D3DX 11 enumerations.

Related topics

[Direct3D 11 Reference](#)

[Direct3D 11 Reference](#)

D3DX Interfaces (Direct3D 11 Graphics)

2/4/2021 • 2 minutes to read • [Edit Online](#)

This section contains reference information for the component object model (COM) interfaces provided by the D3DX utility library.

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

In this section

TOPIC	DESCRIPTION
ID3DX11DataLoader	<p>[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p>Data loading object used by ID3DX11ThreadPump Interface for loading data asynchronously.</p>
ID3DX11DataProcessor	<p>[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p>Data processing object used by ID3DX11ThreadPump Interface for loading data asynchronously.</p>
ID3DX11ThreadPump	<p>[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p>A thread pump executes tasks asynchronously. It is created by calling D3DX11CreateThreadPump. There are several APIs that take an optional thread pump as a parameter, such as D3DX11CreateTextureFromFile and D3DX11CompileFromFile; if you pass a thread pump interface into these APIs, the functions will execute asynchronously on a separate thread. Particularly on multiprocessor machines, a thread pump can load resources and process data without a noticeable decrease in performance.</p>

Related topics

[D3DX 11 Reference](#)

ID3DX11DataLoader interface

2/4/2021 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

Data loading object used by [ID3DX11ThreadPump Interface](#) for loading data asynchronously.

Members

The **ID3DX11DataLoader** interface inherits from the [IUnknown](#) interface. **ID3DX11DataLoader** also has these types of members:

- [Methods](#)

Methods

The **ID3DX11DataLoader** interface has these methods.

METHOD	DESCRIPTION
Decompress	<p>[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p>Decompresses encoded data.</p>
Destroy	<p>[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p>Destroys the loader after a work item completes.</p>
Load	<p>[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p>Loads data from a disk.</p>

Remarks

This object can be inherited and its members redefined. Doing so would enable you to customize the API for

loading and decompressing your own custom file formats.

Requirements

Requirement	Value
Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Header	D3DX11core.h
Library	D3DX11.lib

See also

[D3DX Interfaces](#)

ID3DX11DataLoader::Decompress method

11/2/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

Decompresses encoded data.

Syntax

```
HRESULT Decompress(  
    [out] void    **ppData,  
    [in]  SIZE_T *pcBytes  
) ;
```

Parameters

ppData [out]

Type: **void****

Pointer to the raw data to decompress.

pcBytes [in]

Type: **SIZE_T***

The size of the data pointed to by ppData.

Return value

Type: **HRESULT**

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

Remarks

Use this method to load resources from file systems, such as ZIP files. When loading from an uncompressed resource, the decompression stage does not have to do any work.

[ID3DX11DataLoader Interface](#) can be inherited and its members redefined to support custom file formats.

Requirements

Header	D3DX11core.h
--------	--------------

Library	D3DX11.lib
---------	------------

See also

[ID3DX11DataLoader](#)

[D3DX Interfaces](#)

ID3DX11DataLoader::Destroy method

2/22/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

Destroys the loader after a work item completes.

Syntax

```
HRESULT Destroy();
```

Parameters

This method has no parameters.

Return value

Type: [HRESULT](#)

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

Remarks

This method is used by an [ID3DX11ThreadPump Interface](#).

Requirements

Header	D3DX11core.h
Library	D3DX11.lib

See also

[ID3DX11DataLoader](#)

[D3DX Interfaces](#)

ID3DX11DataLoader::Load method

2/22/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

Loads data from a disk.

Syntax

```
HRESULT Load();
```

Parameters

This method has no parameters.

Return value

Type: [HRESULT](#)

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

Remarks

This method is used by an [ID3DX11ThreadPump Interface](#).

Requirements

Header	D3DX11core.h
Library	D3DX11.lib

See also

[ID3DX11DataLoader](#)

[D3DX Interfaces](#)

ID3DX11DataProcessor interface

2/4/2021 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

Data processing object used by [ID3DX11ThreadPump Interface](#) for loading data asynchronously.

Members

The **ID3DX11DataProcessor** interface inherits from the [IUnknown](#) interface. **ID3DX11DataProcessor** also has these types of members:

- [Methods](#)

Methods

The **ID3DX11DataProcessor** interface has these methods.

METHOD	DESCRIPTION
CreateDeviceObject	<p>[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p>Creates a device object.</p>
Destroy	<p>[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p>Destroys the processor after a work item completes.</p>
Process	<p>[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p>Processes data.</p>

Remarks

This object can be inherited and its members redefined for processing custom file formats.

Requirements

Requirement	Value
Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Header	D3DX11core.h
Library	D3DX11.lib

See also

[D3DX Interfaces](#)

ID3DX11DataProcessor::CreateDeviceObject method

2/22/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

Creates a device object.

Syntax

```
HRESULT CreateDeviceObject(  
    [out] void **ppDataObject  
) ;
```

Parameters

ppDataObject [out]

Type: **void****

Address of a pointer to the created device object.

Return value

Type: **HRESULT**

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

Remarks

This method is used by an [ID3DX11ThreadPump Interface](#).

Requirements

Header	D3DX11core.h
Library	D3DX11.lib

See also

[ID3DX11DataProcessor](#)

[D3DX Interfaces](#)

ID3DX11DataProcessor::Destroy method

2/22/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

Destroys the processor after a work item completes.

Syntax

```
HRESULT Destroy();
```

Parameters

This method has no parameters.

Return value

Type: [HRESULT](#)

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

Remarks

This method is used by an [ID3DX11ThreadPump Interface](#).

Requirements

Header	D3DX11core.h
Library	D3DX11.lib

See also

[ID3DX11DataProcessor](#)

[D3DX Interfaces](#)

ID3DX11DataProcessor::Process method

11/2/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

Processes data.

Syntax

```
HRESULT Process(  
    [in] void    *pData,  
    [in] SIZE_T cBytes  
>;
```

Parameters

pData [in]

Type: **void***

Pointer to the data to be processed.

cBytes [in]

Type: **SIZE_T**

Size of the data to be processed.

Return value

Type: **HRESULT**

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

Remarks

This method is used by an [ID3DX11ThreadPump Interface](#).

Requirements

Header	D3DX11core.h
Library	D3DX11.lib

See also

[ID3DX11DataProcessor](#)

[D3DX Interfaces](#)

ID3DX11ThreadPump interface

2/4/2021 • 3 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

A thread pump executes tasks asynchronously. It is created by calling [D3DX11CreateThreadPump](#). There are several APIs that take an optional thread pump as a parameter, such as [D3DX11CreateTextureFromFile](#) and [D3DX11CompileFromFile](#); if you pass a thread pump interface into these APIs, the functions will execute asynchronously on a separate thread. Particularly on multiprocessor machines, a thread pump can load resources and process data without a noticeable decrease in performance.

Members

The **ID3DX11ThreadPump** interface inherits from the [IUnknown](#) interface. **ID3DX11ThreadPump** also has these types of members:

- [Methods](#)

Methods

The **ID3DX11ThreadPump** interface has these methods.

METHOD	DESCRIPTION
AddWorkItem	<p>[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p>Adds a work item to the thread pump.</p>
GetQueueStatus	<p>[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p>Gets the number of items in each of the three queues inside the thread pump.</p>

METHOD	DESCRIPTION
GetWorkItemCount	<p>[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p>Gets the number of work items in the thread pump.</p>
ProcessDeviceWorkItems	<p>[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p>Sets work items to the device after they have finished loading and processing.</p>
PurgeAllItems	<p>[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p>Clears all work items from the thread pump.</p>
WaitForAllItems	<p>[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p>Waits for all work items in the thread pump to finish.</p>

Remarks

Using a Thread Pump

The thread pump loads and processes data using the following three-step process:

1. Load and decompress the data with a [Data Loader](#). The data loader object has three methods that the thread pump will call internally as it is loading and decompressing the data: [ID3DX11DataLoader::Load](#), [ID3DX11DataLoader::Decompress](#), and [ID3DX11DataLoader::Destroy](#). The specific functionality of these three APIs differs depending on the type of data being loaded and decompressed. The data loader interface can also be inherited and its APIs can be changed if one is loading a data file defined in one's own custom format.
2. Process the data with a [Data Processor](#). The data processor object has three methods that the thread pump will call internally as it is processing the data: [ID3DX11DataProcessor::Process](#), [ID3DX11DataProcessor::CreateDeviceObject](#), and [ID3DX11DataProcessor::Destroy](#). The way it processes the data will be different depending on the type of data. For example, if the data is a texture stored as a JPEG, then [ID3DX11DataProcessor::Process](#) will do JPEG decompression to get the image's raw image bits. If the data is a shader, then [ID3DX11DataProcessor::Process](#) will compile the HLSL into bytecode. After the data has been processed a device object will be created for that data (with

[ID3DX11DataProcessor::CreateDeviceObject](#)) and the object will be added to a queue of device objects. The data processor interface can also be inherited and its APIs can be changed if one is processing a data file defined in one's own custom format.

3. Bind the device object to the device. This is done when one's application calls [ID3DX11ThreadPump::ProcessDeviceWorkItems](#), which will bind a specified number of objects in the queue of device objects to the device.

The thread pump can be used to load data in one of two ways: by calling an API that takes a thread pump as a parameter, such as [D3DX11CreateTextureFromFile](#) and [D3DX11CompileFromFile](#), or by calling

[ID3DX11ThreadPump::AddWorkItem](#). In the case of the APIs that take a thread pump, the data loader and data processor are created internally. In the case of AddWorkItem, the data loader and data processor must be created beforehand and are then passed into AddWorkItem. D3DX11 provides a set of APIs for creating data loaders and data processors that have functionality for loading and processing common data formats. For custom data formats, the data loader and data processor interfaces must be inherited and their methods must be redefined.

The thread pump object takes up a substantial amount of resources, so generally only one should be created per application.

Requirements

Requirement	Value
Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Header	D3DX11core.h
Library	D3DX11.lib

See also

[D3DX Interfaces](#)

ID3DX11ThreadPump::AddWorkItem method

2/22/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

Adds a work item to the thread pump.

Syntax

```
HRESULT AddWorkItem(
    [in] ID3DX11DataLoader     *pDataLoader,
    [in] ID3DX11DataProcessor *pDataProcessor,
    [in] HRESULT              *pHResult,
    [out] void                **ppDeviceObject
);
```

Parameters

pDataLoader [in]

Type: [ID3DX11DataLoader*](#)

The loader that the thread pump will use when a work item requires data to be loaded.

pDataProcessor [in]

Type: [ID3DX11DataProcessor*](#)

The processor that the thread pump will use when a work item requires data to be processed.

pHResult [in]

Type: [HRESULT](#)*

A pointer to the return value. May be **NULL**.

ppDeviceObject [out]

Type: [void**](#)

The device that uses the object.

Return value

Type: [HRESULT](#)

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

Requirements

Header	D3DX11core.h
Library	D3DX11.lib

See also

[ID3DX11ThreadPump](#)

[D3DX Interfaces](#)

ID3DX11ThreadPump::GetQueueStatus method

11/2/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

Gets the number of items in each of the three queues inside the thread pump.

Syntax

```
HRESULT GetQueueStatus(  
    [in] UINT *pIoQueue,  
    [in] UINT *pProcessQueue,  
    [in] UINT *pDeviceQueue  
) ;
```

Parameters

pIoQueue [in]

Type: [UINT*](#)

Number of items in the I/O queue.

pProcessQueue [in]

Type: [UINT*](#)

Number of items in the process queue.

pDeviceQueue [in]

Type: [UINT*](#)

Number of items in the device queue.

Return value

Type: [HRESULT](#)

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

Requirements

Header	D3DX11core.h
--------	--------------

Library	D3DX11.lib
---------	------------

See also

[ID3DX11ThreadPump](#)

[D3DX Interfaces](#)

ID3DX11ThreadPump::GetWorkItemCount method

11/2/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

Gets the number of work items in the thread pump.

Syntax

```
UINT GetWorkItemCount();
```

Parameters

This method has no parameters.

Return value

Type: [UINT](#)

The number of work items queued in the thread pump.

Requirements

Header	D3DX11core.h
Library	D3DX11.lib

See also

[ID3DX11ThreadPump](#)

[D3DX Interfaces](#)

ID3DX11ThreadPump::ProcessDeviceWorkItems method

11/2/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

Sets work items to the device after they have finished loading and processing.

Syntax

```
HRESULT ProcessDeviceWorkItems(  
    [in] UINT iWorkItemCount  
>;
```

Parameters

iWorkItemCount [in]

Type: **UINT**

The number of work items to set to the device.

Return value

Type: **HRESULT**

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

Remarks

When the thread pump has finished loading and processing a resource or shader, it will hold it in a queue until this API is called, at which point the processed items will be set to the device. This is useful for controlling the amount of processing that is spent on binding resources to the device for each frame.

As an example of how one might use this API, say you are nearing the end of one level in your game and you want to begin preloading the textures, shaders, and other resources for the next level. The thread pump will begin loading, decompressing, and processing the resources and shaders on a separate thread until they are ready to be set to the device, at which point it will leave them in a queue. One may not want to set all the resources and shaders to the device at once because this may cause a noticeable temporary slow down in the game's performance. So, this API could be called once per frame so that only a small number work items would be set to the device on each frame, thereby spreading the work load of binding resources to the device over several frames.

Requirements

Header	D3DX11core.h
Library	D3DX11.lib

See also

[ID3DX11ThreadPump](#)

[D3DX Interfaces](#)

ID3DX11ThreadPump::PurgeAllItems method

2/22/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

Clears all work items from the thread pump.

Syntax

```
HRESULT PurgeAllItems();
```

Parameters

This method has no parameters.

Return value

Type: [HRESULT](#)

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

Requirements

Header	D3DX11core.h
Library	D3DX11.lib

See also

[ID3DX11ThreadPump](#)

[D3DX Interfaces](#)

ID3DX11ThreadPump::WaitForAllItems method

2/22/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

Waits for all work items in the thread pump to finish.

Syntax

```
HRESULT WaitForAllItems();
```

Parameters

This method has no parameters.

Return value

Type: [HRESULT](#)

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

Requirements

Header	D3DX11core.h
Library	D3DX11.lib

See also

[ID3DX11ThreadPump](#)

[D3DX Interfaces](#)

D3DX Functions (Direct3D 11 Graphics)

2/4/2021 • 9 minutes to read • [Edit Online](#)

This section contains information about the D3DX 11 functions.

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

In this section

TOPIC	DESCRIPTION
D3DX11CompileFromFile	<p>[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p>Compile a shader or an effect from a file.</p>
D3DX11CompileFromMemory	<p>[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p>Compile a shader or an effect that is loaded in memory.</p>

Topic	Description
D3DX11CompileFromResource	<p data-bbox="853 204 933 233">[!Note]</p> <p data-bbox="853 235 1366 330">The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p data-bbox="853 435 933 464">[!Note]</p> <p data-bbox="853 467 1418 592">Instead of using this function, we recommend that you use resource functions, then compile offline by using the Fxc.exe command-line compiler or use one of the HLSL compile APIs, like the D3DCompile API.</p> <p data-bbox="853 669 1302 698">Compile a shader or an effect from a resource.</p>
D3DX11ComputeNormalMap	<p data-bbox="853 788 933 817">[!Note]</p> <p data-bbox="853 819 1366 914">The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p data-bbox="853 1019 933 1048">[!Note]</p> <p data-bbox="853 1051 1406 1120">Instead of using this function, we recommend that you use the DirectXTex library, ComputeNormalMap.</p> <p data-bbox="853 1197 1366 1291">Converts a height map into a normal map. The (x,y,z) components of each normal are mapped to the (r,g,b) channels of the output texture.</p>
D3DX11CreateAsyncCompilerProcessor	<p data-bbox="853 1356 933 1385">[!Note]</p> <p data-bbox="853 1388 1366 1482">The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps. See Remarks.</p> <p data-bbox="853 1567 1361 1596">Create an asynchronous-data processor for a shader.</p>
D3DX11CreateAsyncFileLoader	<p data-bbox="853 1671 933 1700">[!Note]</p> <p data-bbox="853 1702 1366 1796">The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps. See Remarks.</p> <p data-bbox="853 1879 1190 1909">Create an asynchronous-file loader.</p>

Topic	Description
D3DX11CreateAsyncMemoryLoader	<p>[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps. See Remarks.</p> <p>Create an asynchronous-memory loader.</p>
D3DX11CreateAsyncResourceLoader	<p>[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps. See Remarks.</p> <p>Create an asynchronous-resource loader.</p>
D3DX11CreateAsyncShaderPreprocessProcessor	<p>[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps. See Remarks.</p> <p>Create a data processor for a shader asynchronously.</p>
D3DX11CreateAsyncTextureInfoProcessor	<p>[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps. See Remarks.</p> <p>Create a data processor to be used with a thread pump.</p>
D3DX11CreateAsyncTextureProcessor	<p>[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps. See Remarks.</p> <p>Create a data processor to be used with a thread pump.</p>
D3DX11CreateAsyncShaderResourceViewProcessor	<p>[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps. See Remarks.</p> <p>Create a data processor that will load a resource and then create a shader-resource view for it. Data processors are a component of the asynchronous data loading feature in D3DX11 that uses thread pumps.</p>

Topic	Description
<p>D3DX11CreateShaderResourceViewFromFile</p>	<p>[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p>[!Note] Instead of using this function, we recommend that you use these:</p> <ul style="list-style-type: none"> • DirectXTK library (runtime), CreateXXXTextureFromFile (where XXX is DDS or WIC) • DirectXTex library (tools), LoadFromXXXFile (where XXX is WIC, DDS, or TGA; WIC doesn't support DDS and TGA; D3DX 9 supported TGA as a common art source format for games) then CreateShaderResourceView <p>Create a shader-resource view from a file.</p>
<p>D3DX11CreateShaderResourceViewFromMemory</p>	<p>[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p>[!Note] Instead of using this function, we recommend that you use these:</p> <ul style="list-style-type: none"> • DirectXTK library (runtime), CreateXXXTextureFromMemory (where XXX is DDS or WIC) • DirectXTex library (tools), LoadFromXXXMemory (where XXX is WIC, DDS, or TGA; WIC doesn't support DDS and TGA; D3DX 9 supported TGA as a common art source format for games) then CreateShaderResourceView <p>Create a shader-resource view from a file in memory.</p>

Topic	Description
<p>D3DX11CreateShaderResourceViewFromResource</p>	<p>[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p>[!Note] Instead of using this function, we recommend that you use resource functions, then these:</p> <ul style="list-style-type: none"> • DirectXTK library (runtime), CreateXXXTextureFromMemory (where XXX is DDS or WIC) • DirectXTex library (tools), LoadFromXXXMemory (where XXX is WIC, DDS, or TGA; WIC doesn't support DDS and TGA; D3DX 9 supported TGA as a common art source format for games) then CreateShaderResourceView <p>Create a shader-resource view from a resource.</p>
<p>D3DX11CreateTextureFromFile</p>	<p>[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p>[!Note] Instead of using this function, we recommend that you use these:</p> <ul style="list-style-type: none"> • DirectXTK library (runtime), CreateXXXTextureFromFile (where XXX is DDS or WIC) • DirectXTex library (tools), LoadFromXXXFile (where XXX is WIC, DDS, or TGA; WIC doesn't support DDS and TGA; D3DX 9 supported TGA as a common art source format for games) then CreateTexture <p>Create a texture resource from a file.</p>

Topic	Description
D3DX11CreateTextureFromMemory	<p data-bbox="858 204 937 233">[!Note]</p> <p data-bbox="858 233 1366 330">The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p data-bbox="858 435 937 464">[!Note]</p> <p data-bbox="858 464 1361 525">Instead of using this function, we recommend that you use these:</p> <ul data-bbox="858 548 1387 804" style="list-style-type: none"> <li data-bbox="858 548 1387 644">• DirectXTK library (runtime), CreateXXXTextureFromMemory (where XXX is DDS or WIC) <li data-bbox="858 644 1387 804">• DirectXTex library (tools), LoadFromXXXMemory (where XXX is WIC, DDS, or TGA; WIC doesn't support DDS and TGA; D3DX 9 supported TGA as a common art source format for games) then CreateTexture <p data-bbox="837 884 1382 945">Create a texture resource from a file residing in system memory.</p>
D3DX11CreateTextureFromResource	<p data-bbox="858 1019 937 1048">[!Note]</p> <p data-bbox="858 1048 1366 1145">The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p data-bbox="858 1251 937 1280">[!Note]</p> <p data-bbox="858 1280 1361 1363">Instead of using this function, we recommend that you use resource functions, then these:</p> <ul data-bbox="858 1385 1387 1619" style="list-style-type: none"> <li data-bbox="858 1385 1387 1482">• DirectXTK library (runtime), CreateXXXTextureFromMemory (where XXX is DDS or WIC) <li data-bbox="858 1482 1387 1619">• DirectXTex library (tools), LoadFromXXXMemory (where XXX is WIC, DDS, or TGA; WIC doesn't support DDS and TGA; D3DX 9 supported TGA as a common art source format for games) then CreateTexture <p data-bbox="837 1700 1236 1738">Create a texture from another resource.</p>
D3DX11CreateThreadPump	<p data-bbox="858 1812 937 1841">[!Note]</p> <p data-bbox="858 1841 1366 1938">The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps. See Remarks.</p> <p data-bbox="837 2007 1069 2046">Create a thread pump.</p>

Topic	Description
D3DX11FilterTexture	<p data-bbox="850 197 1429 323">[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p data-bbox="850 428 1429 554">[!Note] Instead of using this function, we recommend that you use the DirectXTex library, GenerateMipMaps and GenerateMipMaps3D.</p> <p data-bbox="850 631 1398 660">Generates mipmap chain using a particular texture filter.</p>
D3DX11GetImageInfoFromFile	<p data-bbox="850 743 1377 869">[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p data-bbox="850 974 1429 1163">[!Note] Instead of using this function, we recommend that you use the DirectXTex library, GetMetadataFromXXXFile (where XXX is WIC, DDS, or TGA; WIC doesn't support DDS and TGA; D3DX 9 supported TGA as a common art source format for games).</p> <p data-bbox="850 1239 1302 1268">Retrieves information about a given image file.</p>
D3DX11GetImageInfoFromMemory	<p data-bbox="850 1356 1377 1459">[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p data-bbox="850 1587 1429 1799">[!Note] Instead of using this function, we recommend that you use the DirectXTex library, GetMetadataFromXXXMemory (where XXX is WIC, DDS, or TGA; WIC doesn't support DDS and TGA; D3DX 9 supported TGA as a common art source format for games).</p> <p data-bbox="850 1875 1358 1933">Get information about an image already loaded into memory.</p>

Topic	Description
D3DX11GetImageInfoFromResource	<p data-bbox="853 204 933 233">[!Note]</p> <p data-bbox="853 235 1366 327">The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p data-bbox="853 435 933 464">[!Note]</p> <p data-bbox="853 467 1418 660">Instead of using this function, we recommend that you use resource functions, then use DirectXTex library (tools), LoadFromXXXMemory (where XXX is WIC, DDS, or TGA; WIC doesn't support DDS and TGA; D3DX 9 supported TGA as a common art source format for games).</p> <p data-bbox="837 727 1398 756">Retrieves information about a given image in a resource.</p>
D3DX11LoadTextureFromTexture	<p data-bbox="853 846 933 875">[!Note]</p> <p data-bbox="853 878 1366 970">The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p data-bbox="853 1075 933 1105">[!Note]</p> <p data-bbox="853 1107 1421 1199">Instead of using this function, we recommend that you use the DirectXTex library, Resize, Convert, Compress, Decompress, and/or CopyRectangle.</p> <p data-bbox="837 1266 1133 1295">Load a texture from a texture.</p>
D3DX11PreprocessShaderFromFile	<p data-bbox="853 1374 933 1403">[!Note]</p> <p data-bbox="853 1405 1366 1498">The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p data-bbox="853 1605 933 1635">[!Note]</p> <p data-bbox="853 1637 1406 1706">Instead of using this function, we recommend that you use the D3DPreprocess API.</p> <p data-bbox="837 1774 1310 1803">Create a shader from a file without compiling it.</p>

Topic	Description
D3DX11PreprocessShaderFromMemory	<p data-bbox="853 204 933 231">[!Note]</p> <p data-bbox="853 233 1366 323">The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p data-bbox="853 428 933 455">[!Note]</p> <p data-bbox="853 458 1402 525">Instead of using this function, we recommend that you use the D3DPreprocess API.</p> <p data-bbox="831 604 1345 631">Create a shader from memory without compiling it.</p>
D3DX11PreprocessShaderFromResource	<p data-bbox="853 709 933 736">[!Note]</p> <p data-bbox="853 738 1366 833">The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p data-bbox="853 938 933 965">[!Note]</p> <p data-bbox="853 968 1402 1035">Instead of using this function, we recommend that you use the D3DPreprocess API.</p> <p data-bbox="831 1114 1366 1140">Create a shader from a resource without compiling it.</p>
D3DX11SaveTextureToFile	<p data-bbox="853 1226 933 1253">[!Note]</p> <p data-bbox="853 1255 1366 1349">The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p data-bbox="853 1455 933 1482">[!Note]</p> <p data-bbox="853 1484 1406 1769">Instead of using this function, we recommend that you use the DirectXTex library, CaptureTexture then SaveToXXXFile (where XXX is WIC, DDS, or TGA; WIC doesn't support DDS and TGA; D3DX 9 supported TGA as a common art source format for games). For the simplified scenario of creating a screen shot from a render target texture, we recommend that you use the DirectXTK library, SaveDDSTextureToFile or SaveWICTextureToFile.</p> <p data-bbox="831 1848 1064 1875">Save a texture to a file.</p>

Topic	Description
D3DX11SaveTextureToMemory	<p data-bbox="842 199 1429 323">[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p data-bbox="842 428 1429 619">[!Note] Instead of using this function, we recommend that you use the DirectXTex library, <code>CaptureTexture</code> then <code>SaveToXXXMemory</code> (where XXX is WIC, DDS, or TGA; WIC doesn't support DDS and TGA; D3DX 9 supported TGA as a common art source format for games).</p> <p data-bbox="842 698 1096 727">Save a texture to memory.</p>
D3DX11SHProjectCubeMap	<p data-bbox="842 806 1366 929">[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p data-bbox="842 1035 1398 1158">[!Note] Instead of using this function, we recommend that you use the Spherical Harmonics Math library, <code>SHProjectCubeMap</code>.</p> <p data-bbox="842 1237 1429 1298">Projects a function represented in a cube map into spherical harmonics.</p>
D3DX11UnsetAllDeviceObjects	<p data-bbox="842 1376 1366 1504">[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p data-bbox="842 1610 1398 1711">[!Note] Instead of using this function, we recommend that you use the ID3D11DeviceContext::ClearState method.</p> <p data-bbox="842 1790 1429 1936">Removes all resources from the device by setting their pointers to NULL. This should be called during shutdown of your application. It helps ensure that when one is releasing all of their resources that none of them are bound to the device.</p>

Related topics

[D3DX 11 Reference](#)

D3DX11CompileFromFile function

11/2/2020 • 3 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

NOTE

Instead of using this function, we recommend that you compile offline by using the Fxc.exe command-line compiler or use one of the HLSL compile APIs, like the [D3DCompileFromFile](#) API.

Compile a shader or an effect from a file.

Syntax

```
HRESULT D3DX11CompileFromFile(
    _In_          LPCTSTR           pSrcFile,
    _In_  const D3D10_SHADER_MACRO *pDefines,
    _In_          LPD3D10INCLUDE    pInclude,
    _In_          LPCSTR            pFunctionName,
    _In_          LPCSTR            pProfile,
    _In_          UINT              Flags1,
    _In_          UINT              Flags2,
    _In_          ID3DX11ThreadPump *pPump,
    _Out_         ID3D10Blob        **ppShader,
    _Out_         ID3D10Blob        **ppErrorMsgs,
    _Out_         HRESULT           *pHRESULT
);
```

Parameters

pSrcFile [in]

Type: [LPCTSTR](#)

The name of the file that contains the shader code. If the compiler settings require Unicode, the data type LPCTSTR resolves to LPCWSTR. Otherwise, the data type resolves to LPCSTR.

pDefines [in]

Type: [const D3D10_SHADER_MACRO*](#)

Optional. Pointer to an array of macro definitions (see [D3D10_SHADER_MACRO](#)). The last structure in the array serves as a terminator and must have all members set to 0. If not used, set *pDefines* to **NULL**.

pInclude [in]

Type: [LPD3D10INCLUDE](#)

Optional. Pointer to an interface for handling include files. Setting this to **NULL** will cause a compile error if a shader contains a #include.

pFunctionName [in]

Type: **LPCSTR**

Name of the shader-entry point function where shader execution begins. When you compile an effect, **D3DX11CompileFromFile** ignores *pFunctionName*; we recommend that you set *pFunctionName* to **NULL** because it is good programming practice to set a pointer parameter to **NULL** if the called function will not use it.

pProfile [in]

Type: **LPCSTR**

A string that specifies the shader model; can be any profile in shader model 2, shader model 3, shader model 4, or shader model 5. The profile can also be for effect type (for example, fx_4_1).

Flags1 [in]

Type: **UINT**

Shader **compile flags**.

Flags2 [in]

Type: **UINT**

Effect **compile flags**. When you compile a shader and not an effect file, **D3DX11CompileFromFile** ignores *Flags2*; we recommend that you set *Flags2* to zero because it is good programming practice to set a nonpointer parameter to zero if the called function will not use it.

pPump [in]

Type: **ID3DX11ThreadPump***

A pointer to a thread pump interface (see [ID3DX11ThreadPump Interface](#)). Use **NULL** to specify that this function should not return until it is completed.

ppShader [out]

Type: **ID3D10Blob****

A pointer to memory which contains the compiled shader, as well as any embedded debug and symbol-table information.

ppErrorMsgs [out]

Type: **ID3D10Blob****

A pointer to memory which contains a listing of errors and warnings that occurred during compilation. These errors and warnings are identical to the debug output from a debugger.

pHResult [out]

Type: **HRESULT***

A pointer to the return value. May be **NULL**. If *pPump* is not **NULL**, then *pHResult* must be a valid memory location until the asynchronous execution completes.

Return value

Type: **HRESULT**

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

D3DX11CompileFromFile returns E_INVALIDARG if you supply a non-NULL value to the *pHResult* parameter when you supply NULL to the *pPump* parameter. For more information about this situation, see Remarks.

Remarks

For more information about **D3DX11CompileFromFile**, see [D3DCompile](#).

You must supply NULL to the *pHResult* parameter if you also supply NULL to the *pPump* parameter. Otherwise, you cannot create a shader by using the compiled shader code that **D3DX11CompileFromFile** returns in the memory that the *ppShader* parameter points to. To create a shader from compiled shader code, you call one of the following [ID3D11Device](#) interface methods:

- [CreateComputeShader](#)
- [CreateDomainShader](#)
- [CreateGeometryShader](#)
- [CreateGeometryShaderWithStreamOutput](#)
- [CreateHullShader](#)
- [CreatePixelShader](#)
- [CreateVertexShader](#)

In addition, if you supply a non-NULL value to *pHResult* when you supply NULL to *pPump*, **D3DX11CompileFromFile** returns the E_INVALIDARG error code.

Requirements

Header	D3DX11async.h
Library	D3DX11.lib

See also

[D3DX Functions](#)

D3DX11CompileFromMemory function

11/2/2020 • 3 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

NOTE

Instead of using this function, we recommend that you compile offline by using the Fxc.exe command-line compiler or use one of the HLSL compile APIs, like the [D3DCompile API](#).

Compile a shader or an effect that is loaded in memory.

Syntax

```
HRESULT D3DX11CompileFromMemory(
    _In_     LPCSTR             pSrcData,
    _In_     SIZE_T              SrcDataLen,
    _In_     LPCSTR             pFileName,
    _In_     const D3D10_SHADER_MACRO *pDefines,
    _In_     LPD3D10INCLUDE      pInclude,
    _In_     LPCSTR             pFunctionName,
    _In_     LPCSTR             pProfile,
    _In_     UINT               Flags1,
    _In_     UINT               Flags2,
    _In_     ID3DX11ThreadPump  *pPump,
    _Out_    ID3D10Blob         **ppShader,
    _Out_    ID3D10Blob         **ppErrorMsgs,
    _Out_    HRESULT            *pHResult
);
```

Parameters

pSrcData [in]

Type: [LPCSTR](#)

Pointer to the shader in memory.

SrcDataLen [in]

Type: [SIZE_T](#)

Size of the shader in memory.

pFileName [in]

Type: [LPCSTR](#)

The name of the file that contains the shader code.

pDefines [in]

Type: **const D3D10_SHADER_MACRO***

Optional. Pointer to an array of macro definitions (see [D3D10_SHADER_MACRO](#)). The last structure in the array serves as a terminator and must have all members set to 0. If not used, set *pDefines* to **NULL**.

pInclude [in]

Type: **LPD3D10INCLUDE**

Optional. Pointer to an interface for handling include files. Setting this to **NULL** will cause a compile error if a shader contains a #include.

pFunctionName [in]

Type: **LPCSTR**

Name of the shader-entry point function where shader execution begins. When you compile an effect, [D3DX11CompileFromMemory](#) ignores *pFunctionName*; we recommend that you set *pFunctionName* to **NULL** because it is good programming practice to set a pointer parameter to **NULL** if the called function will not use it.

pProfile [in]

Type: **LPCSTR**

A string that specifies the shader model; can be any profile in shader model 2, shader model 3, shader model 4, or shader model 5. The profile can also be for effect type (for example, fx_4_1).

Flags1 [in]

Type: **UINT**

Shader [compile flags](#).

Flags2 [in]

Type: **UINT**

Effect [compile flags](#). When you compile a shader and not an effect file, [D3DX11CompileFromMemory](#) ignores *Flags2*; we recommend that you set *Flags2* to zero because it is good programming practice to set a nonpointer parameter to zero if the called function will not use it.

pPump [in]

Type: **ID3DX11ThreadPump***

A pointer to a thread pump interface (see [ID3DX11ThreadPump Interface](#)). Use **NULL** to specify that this function should not return until it is completed.

ppShader [out]

Type: **ID3D10Blob****

A pointer to memory which contains the compiled shader, as well as any embedded debug and symbol-table information.

ppErrorMsgs [out]

Type: **ID3D10Blob****

A pointer to memory which contains a listing of errors and warnings that occurred during compilation. These errors and warnings are identical to the debug output from a debugger.

pHResult [out]

Type: [HRESULT](#)*

A pointer to the return value. May be **NULL**. If *pPump* is not **NULL**, then *pHResult* must be a valid memory location until the asynchronous execution completes.

Return value

Type: [HRESULT](#)

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

D3DX11CompileFromMemory returns **E_INVALIDARG** if you supply non-**NULL** to the *pHResult* parameter when you supply **NULL** to the *pPump* parameter. For more information about this situation, see Remarks.

Remarks

For more information about **D3DX11CompileFromMemory**, see [D3DCompile](#).

You must supply **NULL** to the *pHResult* parameter if you also supply **NULL** to the *pPump* parameter. Otherwise, you cannot subsequently create a shader by using the compiled shader code that

D3DX11CompileFromMemory returns in the memory that the *ppShader* parameter points to. To create a shader from complied shader code, you call one of the following [ID3D11Device](#) interface methods:

- [CreateComputeShader](#)
- [CreateDomainShader](#)
- [CreateGeometryShader](#)
- [CreateGeometryShaderWithStreamOutput](#)
- [CreateHullShader](#)
- [CreatePixelShader](#)
- [CreateVertexShader](#)

In addition, if you supply a non-**NULL** value to *pHResult* when you supply **NULL** to *pPump*, **D3DX11CompileFromMemory** returns the **E_INVALIDARG** error code.

Requirements

Header	D3DX11async.h
Library	D3DX11.lib

See also

[D3DX Functions](#)

D3DX11CompileFromResource function

11/2/2020 • 3 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

NOTE

Instead of using this function, we recommend that you use [resource functions](#), then compile offline by using the Fxc.exe command-line compiler or use one of the HLSL compile APIs, like the [D3DCompile API](#).

Compile a shader or an effect from a resource.

Syntax

```
HRESULT D3DX11CompileFromResource(
    _In_          HMODULE           hSrcModule,
    _In_          LPCTSTR            pSrcResource,
    _In_          LPCTSTR            pSrcFileName,
    _In_  const D3D10_SHADER_MACRO *pDefines,
    _In_          LPD3D10INCLUDE     pInclude,
    _In_          LPCSTR             pFunctionName,
    _In_          LPCSTR             pProfile,
    _In_          UINT               Flags1,
    _In_          UINT               Flags2,
    _In_          ID3DX11ThreadPump *pPump,
    _Out_         ID3D10Blob        **ppShader,
    _Out_         ID3D10Blob        **ppErrorMsgs,
    _Out_         HRESULT            *pHResult
);
```

Parameters

hSrcModule [in]

Type: [HMODULE](#)

Handle to the resource module containing the shader. HMODULE can be obtained with [GetModuleHandle Function](#).

pSrcResource [in]

Type: [LPCTSTR](#)

Name of the resource containing the shader. If the compiler settings require Unicode, the data type LPCTSTR resolves to LPCWSTR. Otherwise, the data type resolves to LPCSTR.

pSrcFileName [in]

Type: [LPCTSTR](#)

Optional. Effect file name, which is used for error messages only. Can be **NULL**.

pDefines [in]

Type: **const D3D10_SHADER_MACRO***

Optional. Pointer to an array of macro definitions (see **D3D10_SHADER_MACRO**). The last structure in the array serves as a terminator and must have all members set to 0. If not used, set *pDefines* to **NULL**.

pInclude [in]

Type: **LPD3D10INCLUDE**

Optional. Pointer to an interface for handling include files. Setting this to **NULL** will cause a compile error if a shader contains a #include.

pFunctionName [in]

Type: **LPCSTR**

Name of the shader-entry point function where shader execution begins. When you compile an effect, **D3DX11CompileFromResource** ignores *pFunctionName*; we recommend that you set *pFunctionName* to **NULL** because it is good programming practice to set a pointer parameter to **NULL** if the called function will not use it.

pProfile [in]

Type: **LPCSTR**

A string that specifies the shader model; can be any profile in shader model 2, shader model 3, shader model 4, or shader model 5. The profile can also be for effect type (for example, fx_4_1).

Flags1 [in]

Type: **UINT**

Shader **compile flags**.

Flags2 [in]

Type: **UINT**

Effect **compile flags**. When you compile a shader and not an effect file, **D3DX11CompileFromResource** ignores *Flags2*; we recommend that you set *Flags2* to zero because it is good programming practice to set a nonpointer parameter to zero if the called function will not use it.

pPump [in]

Type: **ID3DX11ThreadPump***

A pointer to a thread pump interface (see **ID3DX11ThreadPump Interface**). Use **NULL** to specify that this function should not return until it is completed.

ppShader [out]

Type: **ID3D10Blob****

A pointer to memory which contains the compiled shader, as well as any embedded debug and symbol-table information.

ppErrorMsgs [out]

Type: **ID3D10Blob****

A pointer to memory which contains a listing of errors and warnings that occurred during compilation. These errors and warnings are identical to the debug output from a debugger.

pHResult [out]

Type: [HRESULT](#)*

A pointer to the return value. May be **NULL**. If *pPump* is not **NULL**, then *pHResult* must be a valid memory location until the asynchronous execution completes.

Return value

Type: [HRESULT](#)

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

D3DX11CompileFromResource returns **E_INVALIDARG** if you supply non-**NULL** to the *pHResult* parameter when you supply **NULL** to the *pPump* parameter. For more information about this situation, see Remarks.

Remarks

For more information about **D3DX11CompileFromResource**, see [D3DCompile](#).

You must supply **NULL** to the *pHResult* parameter if you also supply **NULL** to the *pPump* parameter. Otherwise, you cannot subsequently create a shader by using the compiled shader code that

D3DX11CompileFromResource returns in the memory that the *ppShader* parameter points to. To create a shader from compiled shader code, you call one of the following [ID3D11Device](#) interface methods:

- [CreateComputeShader](#)
- [CreateDomainShader](#)
- [CreateGeometryShader](#)
- [CreateGeometryShaderWithStreamOutput](#)
- [CreateHullShader](#)
- [CreatePixelShader](#)
- [CreateVertexShader](#)

In addition, if you supply a non-**NULL** value to *pHResult* when you supply **NULL** to *pPump*, **D3DX11CompileFromResource** returns the **E_INVALIDARG** error code.

Requirements

Header	D3DX11async.h
Library	D3DX11.lib

See also

[D3DX Functions](#)

D3DX11ComputeNormalMap function

11/2/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

NOTE

Instead of using this function, we recommend that you use the [DirectXTex](#) library, [ComputeNormalMap](#).

Converts a height map into a normal map. The (x,y,z) components of each normal are mapped to the (r,g,b) channels of the output texture.

Syntax

```
HRESULT D3DX11ComputeNormalMap(
    _In_ ID3D11DeviceContext *pContext,
    _In_ ID3D11Texture2D     *pSrcTexture,
    _In_ UINT                 Flags,
    _In_ UINT                 Channel,
    _In_ FLOAT                Amplitude,
    _In_ ID3D11Texture2D     *pDestTexture
);
```

Parameters

pContext [in]

Type: [ID3D11DeviceContext*](#)

Pointer to an [ID3D11DeviceContext](#) interface, representing the source height-map texture.

pSrcTexture [in]

Type: [ID3D11Texture2D*](#)

Pointer to an [ID3D11Texture2D](#) interface, representing the source height-map texture.

Flags [in]

Type: [UINT](#)

One or more D3DX_NORMALMAP flags that control generation of normal maps.

Channel [in]

Type: [UINT](#)

One D3DX_CHANNEL flag specifying the source of height information.

Amplitude [in]

Type: [FLOAT](#)

Constant value multiplier that increases (or decreases) the values in the normal map. Higher values usually make bumps more visible, lower values usually make bumps less visible.

pDestTexture [in]

Type: [ID3D11Texture2D*](#)

Pointer to an [ID3D11Texture2D](#) interface, representing the destination texture.

Return value

Type: [HRESULT](#)

If the function succeeds, the return value is D3D_OK. If the function fails, the return value can be the following value: D3DERR_INVALIDCALL.

Remarks

This method computes the normal by using the central difference with a kernel size of 3x3. RGB channels in the destination contain biased (x,y,z) components of the normal. The central differencing denominator is hardcoded to 2.0.

Requirements

Header	D3DX11tex.h
Library	D3DX11.lib

See also

[D3DX Functions](#)

D3DX11CreateAsyncCompilerProcessor function

11/2/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps. See Remarks.

Create an asynchronous-data processor for a shader.

Syntax

```
HRESULT D3DX11CreateAsyncCompilerProcessor(
    _In_          LPCSTR             pFileName,
    _In_  const D3D11_SHADER_MACRO   *pDefines,
    _In_          LPD3D10INCLUDE     pInclude,
    _In_          LPCSTR             pFunctionName,
    _In_          LPCSTR             pProfile,
    _In_          UINT               Flags1,
    _In_          UINT               Flags2,
    _Out_         ID3D10Blob        **ppCompiledShader,
    _Out_         ID3D10Blob        **ppErrorBuffer,
    _Out_         ID3DX11DataProcessor **ppDataProcessor
);
```

Parameters

pFileName [in]

Type: [LPCSTR](#)

A string that contains the shader filename.

pDefines [in]

Type: [const D3D11_SHADER_MACRO*](#)

A NULL-terminated array of shader macros; set this to **NULL** to specify no macros.

pInclude [in]

Type: [LPD3D10INCLUDE](#)

A pointer to an include interface. This parameter can be **NULL**.

pFunctionName [in]

Type: [LPCSTR](#)

Name of the shader-entry point function where shader execution begins. When you compile an effect, **D3DX11CreateAsyncCompilerProcessor** ignores *pFunctionName*; we recommend that you set *pFunctionName* to **NULL** because it is good programming practice to set a pointer parameter to **NULL** if the called function will not use it..

pProfile [in]

Type: [LPCSTR](#)

A string that specifies the shader profile or shader model; can be any profile in shader model 2, shader model 3, shader model 4, or shader model 5. The profile can also be for effect type (for example, fx_4_1).

Flags1 [in]

Type: [UINT](#)

Shader compile flags.

Flags2 [in]

Type: [UINT](#)

Effect compile flags. When you compile a shader and not an effect file,

[D3DX11CreateAsyncCompilerProcessor](#) ignores *Flags2*; we recommend that you set *Flags2* to zero because it is good programming practice to set a nonpointer parameter to zero if the called function will not use it.

ppCompiledShader [out]

Type: [ID3D10Blob**](#)

Address of a pointer to the compiled effect.

ppErrorBuffer [out]

Type: [ID3D10Blob**](#)

Address of a pointer to compile errors.

ppDataProcessor [out]

Type: [ID3DX11DataProcessor**](#)

Address of a pointer to a buffer that contains the data processor created (see [ID3DX11DataProcessor Interface](#)).

Return value

Type: [HRESULT](#)

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

Remarks

There is no implementation of the async loader outside of D3DX 10, and D3DX 11.

For Windows Store apps, the DirectX samples (for example, the [Direct3D tutorial sample](#)) include the **BasicLoader** module that uses the Windows Runtime asynchronous programming model ([AsyncBase](#)).

For Win32 desktop apps, you can use the [Concurrency Runtime](#) to implement something similar to the Windows Runtime asynchronous programming model.

Requirements

Header	D3DX11async.h

Library

D3DX11.lib

See also

[D3DX Functions](#)

D3DX11CreateAsyncFileLoader function

11/2/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps. See Remarks.

Create an asynchronous-file loader.

Syntax

```
HRESULT D3DX11CreateAsyncFileLoader(
    _In_     LPCTSTR          pFileName,
    _Out_    ID3DX11DataLoader **ppDataLoader
);
```

Parameters

pFileName [in]

Type: [LPCTSTR](#)

The name of the file to load. If the compiler settings require Unicode, the data type LPCTSTR resolves to LPCWSTR. Otherwise, the data type resolves to LPCSTR.

ppDataLoader [out]

Type: [ID3DX11DataLoader**](#)

Address of a pointer to the asynchronous-data loader (see [ID3DX11DataLoader Interface](#)).

Return value

Type: [HRESULT](#)

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

Remarks

There is no implementation of the async loader outside of D3DX 10, and D3DX 11.

For Windows Store apps, the DirectX samples (for example, the [Direct3D tutorial sample](#)) include the **BasicLoader** module that uses the Windows Runtime asynchronous programming model ([AsyncBase](#)).

For Win32 desktop apps, you can use the [Concurrency Runtime](#) to implement something similar to the Windows Runtime asynchronous programming model.

Requirements

Header	D3DX11async.h
Library	D3DX11.lib

See also

[D3DX Functions](#)

D3DX11CreateAsyncMemoryLoader function

11/2/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps. See Remarks.

Create an asynchronous-memory loader.

Syntax

```
HRESULT D3DX11CreateAsyncMemoryLoader(
    _In_    LPCVOID          pData,
    _In_    SIZE_T           cbData,
    _Out_   ID3DX11DataLoader **ppDataLoader
);
```

Parameters

pData [in]

Type: [LPCVOID](#)

Pointer to the data.

cbData [in]

Type: [SIZE_T](#)

Size of the data.

ppDataLoader [out]

Type: [ID3DX11DataLoader**](#)

The address of a pointer to the asynchronous-data loader (see [ID3DX11DataLoader Interface](#)).

Return value

Type: [HRESULT](#)

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

Remarks

There is no implementation of the async loader outside of D3DX 10, and D3DX 11.

For Windows Store apps, the DirectX samples (for example, the [Direct3D tutorial sample](#)) include the **BasicLoader** module that uses the Windows Runtime asynchronous programming model ([AsyncBase](#)).

For Win32 desktop apps, you can use the [Concurrency Runtime](#) to implement something similar to the Windows Runtime asynchronous programming model.

Requirements

Header	D3DX11async.h
Library	D3DX11.lib

See also

[D3DX Functions](#)

D3DX11CreateAsyncResourceLoader function

11/2/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps. See Remarks.

Create an asynchronous-resource loader.

Syntax

```
HRESULT D3DX11CreateAsyncResourceLoader(
    _In_     HMODULE           hSrcModule,
    _In_     LPCTSTR            pSrcResource,
    _Out_    ID3DX11DataLoader **ppDataLoader
);
```

Parameters

hSrcModule [in]

Type: [HMODULE](#)

Handle to the resource module. Use [GetModuleHandle Function](#) to get the handle.

pSrcResource [in]

Type: [LPCTSTR](#)

Name of the resource in *hSrcModule*. If the compiler settings require Unicode, the data type LPCTSTR resolves to LPCWSTR. Otherwise, the data type resolves to LPCSTR.

ppDataLoader [out]

Type: [ID3DX11DataLoader**](#)

The address of a pointer to the asynchronous-data loader (see [ID3DX11DataLoader Interface](#)).

Return value

Type: [HRESULT](#)

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

Remarks

There is no implementation of the async loader outside of D3DX 10, and D3DX 11.

For Windows Store apps, the DirectX samples (for example, the [Direct3D tutorial sample](#)) include the [BasicLoader](#) module that uses the Windows Runtime asynchronous programming model ([AsyncBase](#)).

For Win32 desktop apps, you can use the [Concurrency Runtime](#) to implement something similar to the

Windows Runtime asynchronous programming model.

Requirements

Header	D3DX11async.h
Library	D3DX11.lib

See also

[D3DX Functions](#)

D3DX11CreateAsyncShaderPreprocessProcessor function

11/2/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps. See Remarks.

Create a data processor for a shader asynchronously.

Syntax

```
HRESULT D3DX11CreateAsyncShaderPreprocessProcessor(
    _In_           LPCSTR             pFileName,
    _In_           const D3D11_SHADER_MACRO *pDefines,
    _In_           LPD3D10INCLUDE     pInclude,
    _Out_          ID3D10Blob        **ppShaderText,
    _Out_          ID3D10Blob        **ppErrorBuffer,
    _Out_          ID3DX11DataProcessor **ppDataProcessor
);
```

Parameters

pFileName [in]

Type: [LPCSTR](#)

A string that contains the shader filename.

pDefines [in]

Type: [const D3D11_SHADER_MACRO*](#)

A NULL-terminated array of shader macros; set this to **NULL** to specify no macros.

pInclude [in]

Type: [LPD3D10INCLUDE](#)

A pointer to an include interface; set this to **NULL** to specify there is no include file.

ppShaderText [out]

Type: [ID3D10Blob**](#)

Address of a pointer to a buffer that contains the ASCII text of the shader.

ppErrorBuffer [out]

Type: [ID3D10Blob**](#)

Address of a pointer to a buffer that contains compile errors.

ppDataProcessor [out]

Type: [ID3DX11DataProcessor**](#)

Address of a pointer to a buffer that contains the data processor created (see [ID3DX11DataProcessor Interface](#)).

Return value

Type: [HRESULT](#)

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

Remarks

There is no implementation of the async loader outside of D3DX 10, and D3DX 11.

For Windows Store apps, the DirectX samples (for example, the [Direct3D tutorial sample](#)) include the **BasicLoader** module that uses the Windows Runtime asynchronous programming model ([AsyncBase](#)).

For Win32 desktop apps, you can use the [Concurrency Runtime](#) to implement something similar to the Windows Runtime asynchronous programming model.

Requirements

Header	D3DX11async.h
Library	D3DX11.lib

See also

[D3DX Functions](#)

D3DX11CreateAsyncTextureInfoProcessor function

11/2/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps. See Remarks.

Create a data processor to be used with a [thread pump](#).

Syntax

```
HRESULT D3DX11CreateAsyncTextureInfoProcessor(
    _In_ D3DX11_IMAGE_INFO    *pImageInfo,
    _Out_ ID3DX11DataProcessor **ppDataProcessor
);
```

Parameters

pImageInfo [in]

Type: [D3DX11_IMAGE_INFO*](#)

Optional. Identifies the characteristics of a texture (see [D3DX11_IMAGE_INFO](#)) when the data processor is created; set this to **NULL** to read the characteristics of a texture when the texture is loaded.

ppDataProcessor [out]

Type: [ID3DX11DataProcessor**](#)

Address of a pointer to a buffer that contains the data processor created (see [ID3DX11DataProcessor Interface](#)).

Return value

Type: [HRESULT](#)

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

Remarks

This API does creates a data-processor interface; [D3DX11CreateAsyncTextureProcessor](#) creates the data-processor interface and loads the texture.

There s no implementation of the async loader outside of D3DX 10, and D3DX 11.

For Windows Store apps, the DirectX samples (for example, the [Direct3D tutorial sample](#)) include the **BasicLoader** module that uses the Windows Runtime asynchronous programming model ([AsyncBase](#)).

For Win32 desktop apps, you can use the [Concurrency Runtime](#) to implement something similar to the Windows Runtime asynchronous programming model.

Requirements

Header	D3DX11tex.h
Library	D3DX11.lib

See also

[D3DX Functions](#)

D3DX11CreateAsyncTextureProcessor function

11/2/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps. See Remarks.

Create a data processor to be used with a [thread pump](#).

Syntax

```
HRESULT D3DX11CreateAsyncTextureProcessor(
    _In_     ID3D11Device             *pDevice,
    _In_     D3DX11_IMAGE_LOAD_INFO  *pLoadInfo,
    _Out_    ID3DX11DataProcessor    **ppDataProcessor
);
```

Parameters

pDevice [in]

Type: [ID3D11Device*](#)

A pointer to the devive (see [ID3D11Device](#)).

pLoadInfo [in]

Type: [D3DX11_IMAGE_LOAD_INFO*](#)

Optional. Identifies the characteristics of a texture (see [D3DX11_IMAGE_LOAD_INFO](#)) when the data processor is created; set this to **NULL** to read the characteristics of a texture when the texture is loaded.

ppDataProcessor [out]

Type: [ID3DX11DataProcessor**](#)

Address of a pointer to a buffer that contains the data processor created (see [ID3DX11DataProcessor Interface](#)).

Return value

Type: [HRESULT](#)

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

Remarks

This API does creates a data-processor interface and loads the texture;
[D3DX11CreateAsyncTextureInfoProcessor](#) creates the data-processor interface.

There s no implementation of the async loader outside of D3DX 10, and D3DX 11.

For Windows Store apps, the DirectX samples (for example, the [Direct3D tutorial sample](#)) include the **BasicLoader** module that uses the Windows Runtime asynchronous programming model ([AsyncBase](#)).

For Win32 desktop apps, you can use the [Concurrency Runtime](#) to implement something similar to the Windows Runtime asynchronous programming model.

Requirements

Header	D3DX11tex.h
Library	D3DX11.lib

See also

[D3DX Functions](#)

D3DX11CreateAsyncShaderResourceViewProcessor function

11/2/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps. See Remarks.

Create a data processor that will load a resource and then create a shader-resource view for it. Data processors are a component of the asynchronous data loading feature in D3DX11 that uses [thread pumps](#).

Syntax

```
HRESULT D3DX11CreateAsyncShaderResourceViewProcessor(
    _In_     ID3D11Device              *pDevice,
    _In_     D3DX11_IMAGE_LOAD_INFO   *pLoadInfo,
    _Out_    ID3DX11DataProcessor    **ppDataProcessor
);
```

Parameters

pDevice [in]

Type: [ID3D11Device*](#)

Pointer to the Direct3D device (see [ID3D11Device](#)) that will be used to create a resource and a shader-resource view for that resource.

pLoadInfo [in]

Type: [D3DX11_IMAGE_LOAD_INFO*](#)

Optional. Identifies the characteristics of a texture (see [D3DX11_IMAGE_LOAD_INFO](#)) when the data processor is created; set this to NULL to read the characteristics of a texture when the texture is loaded.

ppDataProcessor [out]

Type: [ID3DX11DataProcessor**](#)

Address of a pointer to a buffer that contains the data processor created (see [ID3DX11DataProcessor Interface](#)).

Return value

Type: [HRESULT](#)

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

Remarks

There is no implementation of the async loader outside of D3DX 10, and D3DX 11.

For Windows Store apps, the DirectX samples (for example, the [Direct3D tutorial sample](#)) include the **BasicLoader** module that uses the Windows Runtime asynchronous programming model ([AsyncBase](#)).

For Win32 desktop apps, you can use the [Concurrency Runtime](#) to implement something similar to the Windows Runtime asynchronous programming model.

Requirements

Header	D3DX11async.h
Library	D3DX11.lib

See also

[D3DX Functions](#)

D3DX11CreateShaderResourceViewFromFile function

11/2/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

NOTE

Instead of using this function, we recommend that you use these:

- [DirectXTK](#) library (runtime), [CreateXXXTextureFromFile](#) (where XXX is DDS or WIC)
- [DirectXTex](#) library (tools), [LoadFromXXXFile](#) (where XXX is WIC, DDS, or TGA; WIC doesn't support DDS and TGA; D3DX 9 supported TGA as a common art source format for games) then [CreateShaderResourceView](#)

Create a shader-resource view from a file.

Syntax

```
HRESULT D3DX11CreateShaderResourceViewFromFile(
    _In_     ID3D11Device                  *pDevice,
    _In_     LPCTSTR                      pSrcFile,
    _In_     D3DX11_IMAGE_LOAD_INFO      *pCreateInfo,
    _In_     ID3DX11ThreadPump          *pPump,
    _Out_    ID3D11ShaderResourceView **ppShaderResourceView,
    _Out_    HRESULT                     *pHResult
);
```

Parameters

pDevice [in]

Type: [ID3D11Device](#)*

A pointer to the device (see [ID3D11Device](#)) that will use the resource.

pSrcFile [in]

Type: [LPCTSTR](#)

Name of the file that contains the shader-resource view. If the compiler settings require Unicode, the data type LPCTSTR resolves to LPCWSTR. Otherwise, the data type resolves to LPCSTR.

pCreateInfo [in]

Type: [D3DX11_IMAGE_LOAD_INFO](#)*

Optional. Identifies the characteristics of a texture (see [D3DX11_IMAGE_LOAD_INFO](#)) when the data processor is created; set this to **NULL** to read the characteristics of a texture when the texture is loaded.

pPump [in]

Type: [ID3DX11ThreadPump*](#)

Pointer to a thread-pump interface (see [ID3DX11ThreadPump Interface](#)). If **NULL** is specified, this function will behave synchronously and will not return until it is finished.

ppShaderResourceView [out]

Type: [ID3D11ShaderResourceView**](#)

Address of a pointer to the shader-resource view (see [ID3D11ShaderResourceView](#)).

pHResult [out]

Type: [HRESULT*](#)

A pointer to the return value. May be **NULL**. If *pPump* is not **NULL**, then *pHResult* must be a valid memory location until the asynchronous execution completes.

Return value

Type: [HRESULT](#)

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

Remarks

For a list of supported image formats, see [D3DX11_IMAGE_FILE_FORMAT](#).

Requirements

Header	D3DX11tex.h
Library	D3DX11.lib

See also

[D3DX Functions](#)

D3DX11CreateShaderResourceViewFromMemory function

11/2/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

NOTE

Instead of using this function, we recommend that you use these:

- [DirectXTK](#) library (runtime), [CreateXXXTextureFromMemory](#) (where XXX is DDS or WIC)
- [DirectXTex](#) library (tools), [LoadFromXXXMemory](#) (where XXX is WIC, DDS, or TGA; WIC doesn't support DDS and TGA; D3DX 9 supported TGA as a common art source format for games) then [CreateShaderResourceView](#)

Create a shader-resource view from a file in memory.

Syntax

```
HRESULT D3DX11CreateShaderResourceViewFromMemory(
    _In_     ID3D11Device                  *pDevice,
    _In_     LPCVOID                      pSrcData,
    _In_     SIZE_T                       SrcDataSize,
    _In_     D3DX11_IMAGE_LOAD_INFO      *pCreateInfo,
    _In_     ID3DX11ThreadPump          *pPump,
    _Out_    ID3D11ShaderResourceView **ppShaderResourceView,
    _Out_    HRESULT                     *pHResult
);
```

Parameters

pDevice [in]

Type: [ID3D11Device*](#)

A pointer to the device (see [ID3D11Device](#)) that will use the resource.

pSrcData [in]

Type: [LPCVOID](#)

Pointer to the file in memory that contains the shader-resource view.

SrcDataSize [in]

Type: [SIZE_T](#)

Size of the file in memory.

pCreateInfo [in]

Type: [D3DX11_IMAGE_LOAD_INFO](#)*

Optional. Identifies the characteristics of a texture (see [D3DX11_IMAGE_LOAD_INFO](#)) when the data processor is created; set this to **NULL** to read the characteristics of a texture when the texture is loaded.

pPump [in]

Type: [ID3DX11ThreadPump](#)*

A pointer to a thread pump interface (see [ID3DX11ThreadPump Interface](#)). If **NULL** is specified, this function will behave synchronously and will not return until it is finished.

ppShaderResourceView [out]

Type: [ID3D11ShaderResourceView](#)**

Address of a pointer to the newly created shader resource view. See [ID3D11ShaderResourceView](#).

pHResult [out]

Type: [HRESULT](#)*

A pointer to the return value. May be **NULL**. If *pPump* is not **NULL**, then *pHResult* must be a valid memory location until the asynchronous execution completes.

Return value

Type: [HRESULT](#)

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

Requirements

Header	D3DX11tex.h
Library	D3DX11.lib

See also

[D3DX Functions](#)

D3DX11CreateShaderResourceViewFromResource function

11/2/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

NOTE

Instead of using this function, we recommend that you use [resource functions](#), then these:

- [DirectXTK](#) library (runtime), [CreateXXXTextureFromMemory](#) (where XXX is DDS or WIC)
- [DirectXTex](#) library (tools), [LoadFromXXXMemory](#) (where XXX is WIC, DDS, or TGA; WIC doesn't support DDS and TGA; D3DX 9 supported TGA as a common art source format for games) then [CreateShaderResourceView](#)

Create a shader-resource view from a resource.

Syntax

```
HRESULT D3DX11CreateShaderResourceViewFromResource(
    _In_     ID3D11Device              *pDevice,
    _In_     HMODULE                  hSrcModule,
    _In_     LPCTSTR                  pSrcResource,
    _In_     D3DX11_IMAGE_LOAD_INFO   *pCreateInfo,
    _In_     ID3DX11ThreadPump        *pPump,
    _Out_    ID3D11ShaderResourceView **ppShaderResourceView,
    _Out_    HRESULT                 *pHResult
);
```

Parameters

pDevice [in]

Type: [ID3D11Device*](#)

A pointer to the device (see [ID3D11Device](#)) that will use the resource.

hSrcModule [in]

Type: [HMODULE](#)

Handle to the resource module containing the shader-resource view. HMODULE can be obtained with [GetModuleHandle Function](#).

pSrcResource [in]

Type: [LPCTSTR](#)

Name of the shader resource view in *hSrcModule*. If the compiler settings require Unicode, the data type

LPCTSTR resolves to LPCWSTR. Otherwise, the data type resolves to LPCSTR.

pLoadInfo [in]

Type: [D3DX11_IMAGE_LOAD_INFO*](#)

Optional. Identifies the characteristics of a texture (see [D3DX11_IMAGE_LOAD_INFO](#)) when the data processor is created; set this to **NULL** to read the characteristics of a texture when the texture is loaded.

pPump [in]

Type: [ID3DX11ThreadPump*](#)

A pointer to a thread pump interface (see [ID3DX11ThreadPump Interface](#)). If **NULL** is specified, this function will behave synchronously and will not return until it is finished.

ppShaderResourceView [out]

Type: [ID3D11ShaderResourceView**](#)

Address of a pointer to the shader-resource view (see [ID3D11ShaderResourceView](#)).

pHResult [out]

Type: [HRESULT*](#)

A pointer to the return value. May be **NULL**. If *pPump* is not **NULL**, then *pHResult* must be a valid memory location until the asynchronous execution completes.

Return value

Type: [HRESULT](#)

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

Requirements

Header	D3DX11tex.h
Library	D3DX11.lib

See also

[D3DX Functions](#)

D3DX11CreateTextureFromFile function

11/2/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

NOTE

Instead of using this function, we recommend that you use these:

- [DirectXTK](#) library (runtime), [CreateXXXTextureFromFile](#) (where XXX is DDS or WIC)
- [DirectXTex](#) library (tools), [LoadFromXXXFile](#) (where XXX is WIC, DDS, or TGA; WIC doesn't support DDS and TGA; D3DX 9 supported TGA as a common art source format for games) then [CreateTexture](#)

Create a texture resource from a file.

Syntax

```
HRESULT D3DX11CreateTextureFromFile(
    _In_     ID3D11Device             *pDevice,
    _In_     LPCTSTR                 pSrcFile,
    _In_     D3DX11_IMAGE_LOAD_INFO  *pCreateInfo,
    _In_     ID3DX11ThreadPump       *pPump,
    _Out_    ID3D11Resource          **ppTexture,
    _Out_    HRESULT                 *pHResult
);
```

Parameters

pDevice [in]

Type: [ID3D11Device](#)*

A pointer to the device (see [ID3D11Device](#)) that will use the resource.

pSrcFile [in]

Type: [LPCTSTR](#)

The name of the file containing the resource. If the compiler settings require Unicode, the data type LPCTSTR resolves to LPCWSTR. Otherwise, the data type resolves to LPCSTR.

pCreateInfo [in]

Type: [D3DX11_IMAGE_LOAD_INFO](#)*

Optional. Identifies the characteristics of a texture (see [D3DX11_IMAGE_LOAD_INFO](#)) when the data processor is created; set this to **NULL** to read the characteristics of a texture when the texture is loaded.

pPump [in]

Type: [ID3DX11ThreadPump](#)*

A pointer to a thread pump interface (see [ID3DX11ThreadPump Interface](#)). If **NULL** is specified, this function will behave synchronously and will not return until it is finished.

ppTexture [out]

Type: [ID3D11Resource](#)**

The address of a pointer to the texture resource (see [ID3D11Resource](#)).

pHResult [out]

Type: [HRESULT](#)*

A pointer to the return value. May be **NULL**. If *pPump* is not **NULL**, then *pHResult* must be a valid memory location until the asynchronous execution completes.

Return value

Type: [HRESULT](#)

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

Requirements

Header	D3DX11.h
Library	D3DX11.lib

See also

[D3DX Functions](#)

D3DX11CreateTextureFromMemory function

11/2/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

NOTE

Instead of using this function, we recommend that you use these:

- [DirectXTK](#) library (runtime), [CreateXXXTextureFromMemory](#) (where XXX is DDS or WIC)
- [DirectXTex](#) library (tools), [LoadFromXXXMemory](#) (where XXX is WIC, DDS, or TGA; WIC doesn't support DDS and TGA; D3DX 9 supported TGA as a common art source format for games) then [CreateTexture](#)

Create a texture resource from a file residing in system memory.

Syntax

```
HRESULT D3DX11CreateTextureFromMemory(
    _In_     ID3D11Device             *pDevice,
    _In_     LPCVOID                 pSrcData,
    _In_     SIZE_T                  SrcDataSize,
    _In_     D3DX11_IMAGE_LOAD_INFO *pCreateInfo,
    _In_     ID3DX11ThreadPump       *pPump,
    _Out_    ID3D11Resource          **ppTexture,
    _Out_    HRESULT                 *pHResult
);
```

Parameters

pDevice [in]

Type: [ID3D11Device](#)*

A pointer to the device (see [ID3D11Device](#)) that will use the resource.

pSrcData [in]

Type: [LPCVOID](#)

Pointer to the resource in system memory.

SrcDataSize [in]

Type: [SIZE_T](#)

Size of the resource in system memory.

pCreateInfo [in]

Type: [D3DX11_IMAGE_LOAD_INFO](#)*

Optional. Identifies the characteristics of a texture (see [D3DX11_IMAGE_LOAD_INFO](#)) when the data processor is created; set this to **NULL** to read the characteristics of a texture when the texture is loaded.

pPump [in]

Type: [ID3DX11ThreadPump*](#)

A pointer to a thread pump interface (see [ID3DX11ThreadPump Interface](#)). If **NULL** is specified, this function will behave synchronously and will not return until it is finished.

ppTexture [out]

Type: [ID3D11Resource**](#)

Address of a pointer to the created resource. See [ID3D11Resource](#).

pHResult [out]

Type: [HRESULT*](#)

A pointer to the return value. May be **NULL**. If *pPump* is not **NULL**, then *pHResult* must be a valid memory location until the asynchronous execution completes.

Return value

Type: [HRESULT](#)

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

Requirements

Header	D3DX11.h
Library	D3DX11.lib

See also

[D3DX Functions](#)

D3DX11CreateTextureFromResource function

11/2/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

NOTE

Instead of using this function, we recommend that you use [resource functions](#), then these:

- [DirectXTK](#) library (runtime), **CreateXXXTextureFromMemory** (where XXX is DDS or WIC)
- [DirectXTex](#) library (tools), **LoadFromXXXMemory** (where XXX is WIC, DDS, or TGA; WIC doesn't support DDS and TGA; D3DX 9 supported TGA as a common art source format for games) then **CreateTexture**

Create a texture from another resource.

Syntax

```
HRESULT D3DX11CreateTextureFromResource(
    _In_     ID3D11Device             *pDevice,
    _In_     HMODULE                 hSrcModule,
    _In_     LPCTSTR                  pSrcResource,
    _In_     D3DX11_IMAGE_LOAD_INFO  *pCreateInfo,
    _In_     ID3DX11ThreadPump       *pPump,
    _Out_    ID3D11Resource          **ppTexture,
    _Out_    HRESULT                 *pHResult
);
```

Parameters

pDevice [in]

Type: [ID3D11Device](#)*

A pointer to the device (see [ID3D11Device](#)) that will use the resource.

hSrcModule [in]

Type: [HMODULE](#)

A handle to the source resource. HMODULE can be obtained with [GetModuleHandle Function](#).

pSrcResource [in]

Type: [LPCTSTR](#)

A string that contains the name of the source resource. If the compiler settings require Unicode, the data type LPCTSTR resolves to LPCWSTR. Otherwise, the data type resolves to LPCSTR.

pCreateInfo [in]

Type: [D3DX11_IMAGE_LOAD_INFO*](#)

Optional. Identifies the characteristics of a texture (see [D3DX11_IMAGE_LOAD_INFO](#)) when the data processor is created; set this to **NULL** to read the characteristics of a texture when the texture is loaded.

pPump [in]

Type: [ID3DX11ThreadPump*](#)

A pointer to a thread pump interface (see [ID3DX11ThreadPump Interface](#)). If **NULL** is specified, this function will behave synchronously and will not return until it is finished.

ppTexture [out]

Type: [ID3D11Resource**](#)

The address of a pointer to the texture resource (see [ID3D11Resource](#)).

pHResult [out]

Type: [HRESULT*](#)

A pointer to the return value. May be **NULL**. If *pPump* is not **NULL**, then *pHResult* must be a valid memory location until the asynchronous execution completes.

Return value

Type: [HRESULT](#)

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

Requirements

Header	D3DX11.h
Library	D3DX11.lib

See also

[D3DX Functions](#)

D3DX11CreateThreadPump function

11/2/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps. See Remarks.

Create a thread pump.

Syntax

```
HRESULT D3DX11CreateThreadPump(
    _In_     UINT           cIoThreads,
    _In_     UINT           cProcThreads,
    _Out_    ID3DX11ThreadPump **ppThreadPump
);
```

Parameters

cIoThreads [in]

Type: **UINT**

The number of I/O threads to create. If 0 is specified, Direct3D will attempt to calculate the optimal number of threads based on the hardware configuration.

cProcThreads [in]

Type: **UINT**

The number of process threads to create. If 0 is specified, Direct3D will attempt to calculate the optimal number of threads based on the hardware configuration.

ppThreadPump [out]

Type: **ID3DX11ThreadPump****

The created thread pump. See [ID3DX11ThreadPump Interface](#).

Return value

Type: **HRESULT**

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

Remarks

A thread pump is a very resource-intensive object. Only one thread pump should be created per application.

There is no implementation of the async loader outside of D3DX 10, and D3DX 11.

For Windows Store apps, the DirectX samples (for example, the [Direct3D tutorial sample](#)) include the

BasicLoader module that uses the Windows Runtime asynchronous programming model ([AsyncBase](#)).

For Win32 desktop apps, you can use the [Concurrency Runtime](#) to implement something similar to the Windows Runtime asynchronous programming model.

Requirements

Header	D3DX11core.h
Library	D3DX11.lib

See also

[D3DX Functions](#)

D3DX11FilterTexture function

11/2/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

NOTE

Instead of using this function, we recommend that you use the [DirectXtex](#) library, [GenerateMipMaps](#) and [GenerateMipMaps3D](#).

Generates mipmap chain using a particular texture filter.

Syntax

```
HRESULT D3DX11FilterTexture(  
    ID3D11DeviceContext *pContext,  
    ID3D11Resource     *pTexture,  
    UINT               SrcLevel,  
    UINT               MipFilter  
)
```

Parameters

pContext

Type: [ID3D11DeviceContext*](#)

A pointer to an [ID3D11DeviceContext](#) object.

pTexture

Type: [ID3D11Resource*](#)

The texture object to be filtered. See [ID3D11Resource](#).

SrcLevel

Type: [UINT](#)

The mipmap level whose data is used to generate the rest of the mipmap chain.

MipFilter

Type: [UINT](#)

Flags controlling how each miplevel is filtered (or D3DX11_DEFAULT for D3DX11_FILTER_LINEAR). See [D3DX11_FILTER_FLAG](#).

Return value

Type: [HRESULT](#)

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

Requirements

Header	D3DX11tex.h
Library	D3DX11.lib

See also

[D3DX Functions](#)

D3DX11GetImageInfoFromFile function

11/2/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

NOTE

Instead of using this function, we recommend that you use the [DirectXTex](#) library, **GetMetadataFromXXXFile** (where XXX is WIC, DDS, or TGA; WIC doesn't support DDS and TGA; D3DX 9 supported TGA as a common art source format for games).

Retrieves information about a given image file.

Syntax

```
HRESULT D3DX11GetImageInfoFromFile(
    _In_     LPCTSTR             pSrcFile,
    _In_     ID3DX11ThreadPump *pPump,
    _In_     D3DX11_IMAGE_INFO *pSrcInfo,
    _Out_    HRESULT             *pHResult
);
```

Parameters

pSrcFile [in]

Type: [LPCTSTR](#)

File name of image to retrieve information about. If UNICODE or _UNICODE are defined, this parameter type is LPCWSTR, otherwise, the type is LPCSTR.

pPump [in]

Type: [ID3DX11ThreadPump*](#)

Optional thread pump that can be used to load the info asynchronously. Can be NULL. See [ID3DX11ThreadPump Interface](#).

pSrcInfo [in]

Type: [D3DX11_IMAGE_INFO*](#)

Pointer to a [D3DX11_IMAGE_INFO](#) to be filled with the description of the data in the source file.

pHResult [out]

Type: [HRESULT](#)*

A pointer to the return value. May be NULL. If *pPump* is not NULL, then *pHResult* must be a valid memory

location until the asynchronous execution completes.

Return value

Type: [HRESULT](#)

If the function succeeds, the return value is D3D_OK. If the function fails, the return value can be the following:
D3DERR_INVALIDCALL

Remarks

This function supports both Unicode and ANSI strings.

Requirements

Header	D3DX11tex.h
Library	D3DX11.lib

See also

[D3DX Functions](#)

D3DX11GetImageInfoFromMemory function

11/2/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

NOTE

Instead of using this function, we recommend that you use the [DirectXTex](#) library, [GetMetadataFromXXXMemory](#) (where XXX is WIC, DDS, or TGA; WIC doesn't support DDS and TGA; D3DX 9 supported TGA as a common art source format for games).

Get information about an image already loaded into memory.

Syntax

```
HRESULT D3DX11GetImageInfoFromMemory(
    _In_    LPCVOID           pSrcData,
    _In_    SIZE_T            SrcDataSize,
    _In_    ID3DX11ThreadPump *pPump,
    _In_    D3DX11_IMAGE_INFO *pSrcInfo,
    _Out_   HRESULT           *pHResult
);
```

Parameters

pSrcData [in]

Type: [LPCVOID](#)

Pointer to the image in memory.

SrcDataSize [in]

Type: [SIZE_T](#)

Size of the image in memory, in bytes.

pPump [in]

Type: [ID3DX11ThreadPump*](#)

Optional thread pump that can be used to load the info asynchronously. Can be **NULL**. See [ID3DX11ThreadPump Interface](#).

pSrcInfo [in]

Type: [D3DX11_IMAGE_INFO*](#)

Information about the image in memory.

pHResult [out]

Type: **HRESULT***

A pointer to the return value. May be **NULL**. If *pPump* is not **NULL**, then *pHResult* must be a valid memory location until the asynchronous execution completes.

Return value

Type: **HRESULT**

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

Requirements

Header	D3DX11tex.h
Library	D3DX11.lib

See also

[D3DX Functions](#)

D3DX11GetImageInfoFromResource function

11/2/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

NOTE

Instead of using this function, we recommend that you use [resource functions](#), then use [DirectXTex](#) library (tools), [LoadFromXXXMemory](#) (where XXX is WIC, DDS, or TGA; WIC doesn't support DDS and TGA; D3DX 9 supported TGA as a common art source format for games).

Retrieves information about a given image in a resource.

Syntax

```
HRESULT D3DX11GetImageInfoFromResource(
    _In_     HMODULE           hSrcModule,
    _In_     LPCTSTR            pSrcResource,
    _In_     ID3DX11ThreadPump *pPump,
    _In_     D3DX11_IMAGE_INFO *pSrcInfo,
    _Out_    HRESULT            *pHResult
);
```

Parameters

hSrcModule [in]

Type: [HMODULE](#)

Module where the resource is loaded. Set this parameter to **NULL** to specify the module associated with the image that the operating system used to create the current process.

pSrcResource [in]

Type: [LPCTSTR](#)

Pointer to a string that specifies the filename. If the compiler settings require Unicode, the data type [LPCTSTR](#) resolves to [LPCWSTR](#). Otherwise, the data type resolves to [LPCSTR](#). See Remarks.

pPump [in]

Type: [ID3DX11ThreadPump*](#)

Optional thread pump that can be used to load the info asynchronously. Can be **NULL**. See [ID3DX11ThreadPump Interface](#).

pSrcInfo [in]

Type: [D3DX11_IMAGE_INFO*](#)

Pointer to a D3DX11_IMAGE_INFO structure to be filled with the description of the data in the source file.

pHResult [out]

Type: **HRESULT***

A pointer to the return value. May be **NULL**. If *pPump* is not **NULL**, then *pHResult* must be a valid memory location until the asynchronous execution completes.

Return value

Type: **HRESULT**

If the function succeeds, the return value is **D3D_OK**. If the function fails, the return value can be the following:
D3DERR_INVALIDCALL

Remarks

The compiler setting also determines the function version. If Unicode is defined, the function call resolves to **D3DX11GetImageInfoFromResourceW**. Otherwise, the function call resolves to **D3DX11GetImageInfoFromResourceA** because ANSI strings are being used.

Requirements

Header	D3DX11tex.h
Library	D3DX11.lib

See also

[D3DX Functions](#)

D3DX11LoadTextureFromTexture function

2/22/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

NOTE

Instead of using this function, we recommend that you use the [DirectXTex](#) library, [Resize](#), [Convert](#), [Compress](#), [Decompress](#), and/or [CopyRectangle](#).

Load a texture from a texture.

Syntax

```
HRESULT D3DX11LoadTextureFromTexture(
    ID3D11DeviceContext      *pContext,
    ID3D11Resource           *pSrcTexture,
    D3DX11_TEXTURE_LOAD_INFO *pLoadInfo,
    ID3D11Resource           *pDstTexture
);
```

Parameters

pContext

Type: [ID3D11DeviceContext*](#)

A pointer to an [ID3D11DeviceContext](#) object.

pSrcTexture

Type: [ID3D11Resource*](#)

Pointer to the source texture. See [ID3D11Resource](#).

pLoadInfo

Type: [D3DX11_TEXTURE_LOAD_INFO*](#)

Pointer to texture loading parameters. See [D3DX11_TEXTURE_LOAD_INFO](#).

pDstTexture

Type: [ID3D11Resource*](#)

Pointer to the destination texture. See [ID3D11Resource](#).

Return value

Type: [HRESULT](#)

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

Requirements

Header	D3DX11tex.h
Library	D3DX11.lib

See also

[D3DX Functions](#)

D3DX11PreprocessShaderFromFile function

11/2/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

NOTE

Instead of using this function, we recommend that you use the [D3DPreprocess](#) API.

Create a shader from a file without compiling it.

Syntax

```
HRESULT D3DX11PreprocessShaderFromFile(
    _In_          LPCTSTR             pFileName,
    _In_          const D3D11_SHADER_MACRO *pDefines,
    _In_          LPD3D10INCLUDE      pInclude,
    _In_          ID3DX11ThreadPump  *pPump,
    _Out_         ID3D10Blob        **ppShaderText,
    _Out_         ID3D10Blob        **ppErrorMsgs,
    _Out_         HRESULT            *pHRESULT
);
```

Parameters

pFileName [in]

Type: [LPCTSTR](#)

Name of the shader file.

pDefines [in]

Type: [const D3D11_SHADER_MACRO*](#)

A NULL-terminated array of shader macros; set this to **NULL** to specify no macros.

pInclude [in]

Type: [LPD3D10INCLUDE](#)

A pointer to an include interface; set this to **NULL** to specify there is no include file.

pPump [in]

Type: [ID3DX11ThreadPump*](#)

A pointer to a thread pump interface (see [ID3DX11ThreadPump Interface](#)). Use **NULL** to specify that this function should not return until it is completed.

ppShaderText [out]

Type: [ID3D10Blob**](#)

A pointer to memory that contains the uncompiled shader.

ppErrorMsgs [out]

Type: [ID3D10Blob**](#)

The address of a pointer to memory that contains effect-creation errors, if any occurred.

pHResult [out]

Type: [HRESULT*](#)

A pointer to the return value. May be **NULL**. If *pPump* is not **NULL**, then *pHResult* must be a valid memory location until the asynchronous execution completes.

Return value

Type: [HRESULT](#)

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

Requirements

Header	D3DX11async.h
Library	D3DX11.lib

See also

[D3DX Functions](#)

D3DX11PreprocessShaderFromMemory function

11/2/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

NOTE

Instead of using this function, we recommend that you use the [D3DPreprocess](#) API.

Create a shader from memory without compiling it.

Syntax

```
HRESULT D3DX11PreprocessShaderFromMemory(
    _In_          LPCSTR           pSrcData,
    _In_          SIZE_T            SrcDataSize,
    _In_          LPCSTR           pFileName,
    _In_          const D3D11_SHADER_MACRO *pDefines,
    _In_          LPD3D10INCLUDE    pInclude,
    _In_          ID3DX11ThreadPump *pPump,
    _Out_         ID3D10Blob       **ppShaderText,
    _Out_         ID3D10Blob       **ppErrorMsgs,
    _Out_         HRESULT          *pHRESULT
);
```

Parameters

pSrcData [in]

Type: [LPCSTR](#)

Pointer to the memory containing the shader.

SrcDataSize [in]

Type: [SIZE_T](#)

Size of the shader.

pFileName [in]

Type: [LPCSTR](#)

Name of the shader.

pDefines [in]

Type: [const D3D11_SHADER_MACRO*](#)

A NULL-terminated array of shader macros; set this to **NULL** to specify no macros.

pInclude [in]

Type: [LPD3D10INCLUDE](#)

A pointer to an include interface; set this to **NULL** to specify there is no include file.

pPump [in]

Type: [ID3DX11ThreadPump*](#)

A pointer to a thread pump interface (see [ID3DX11ThreadPump Interface](#)). Use **NULL** to specify that this function should not return until it is completed.

ppShaderText [out]

Type: [ID3D10Blob**](#)

A pointer to memory that contains the uncompiled shader.

ppErrorMsgs [out]

Type: [ID3D10Blob**](#)

The address of a pointer to memory that contains effect-creation errors, if any occurred.

pHResult [out]

Type: [HRESULT*](#)

A pointer to the return value. May be **NULL**. If *pPump* is not **NULL**, then *pHResult* must be a valid memory location until the asynchronous execution completes.

Return value

Type: [HRESULT](#)

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

Requirements

Header	D3DX11async.h
Library	D3DX11.lib

See also

[D3DX Functions](#)

D3DX11PreprocessShaderFromResource function

11/2/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

NOTE

Instead of using this function, we recommend that you use the [D3DPreprocess](#) API.

Create a shader from a resource without compiling it.

Syntax

```
HRESULT D3DX11PreprocessShaderFromResource(
    _In_          HMODULE           hModule,
    _In_          LPCTSTR            pResourceName,
    _In_          LPCTSTR            pSrcFileName,
    _In_          const D3D11_SHADER_MACRO *pDefines,
    _In_          LPD3D10INCLUDE      pInclude,
    _In_          ID3DX11ThreadPump  *pPump,
    _Out_         ID3D10Blob        **ppShaderText,
    _Out_         ID3D10Blob        **ppErrorMsgs,
    _Out_         HRESULT            *pHRESULT
);
```

Parameters

hModule [in]

Type: [HMODULE](#)

Handle to the resource module containing the shader. HMODULE can be obtained with [GetModuleHandle Function](#).

pResourceName [in]

Type: [LPCTSTR](#)

The name of the resource in side *hModule* containing the shader. If the compiler settings require Unicode, the data type LPCTSTR resolves to LPCWSTR. Otherwise, the data type resolves to LPCSTR.

pSrcFileName [in]

Type: [LPCTSTR](#)

Optional. Effect file name, which is used for error messages only. Can be **NULL**.

pDefines [in]

Type: [const D3D11_SHADER_MACRO*](#)

A NULL-terminated array of shader macros; set this to **NULL** to specify no macros.

pInclude [in]

Type: [LPD3D10INCLUDE](#)

A pointer to an include interface; set this to **NULL** to specify there is no include file.

pPump [in]

Type: [ID3DX11ThreadPump*](#)

A pointer to a thread pump interface (see [ID3DX11ThreadPump Interface](#)). Use **NULL** to specify that this function should not return until it is completed.

ppShaderText [out]

Type: [ID3D10Blob**](#)

A pointer to memory that contains the uncompiled shader.

ppErrorMsgs [out]

Type: [ID3D10Blob**](#)

The address of a pointer to memory that contains effect-creation errors, if any occurred.

pHResult [out]

Type: [HRESULT*](#)

A pointer to the return value. May be **NULL**. If *pPump* is not **NULL**, then *pHResult* must be a valid memory location until the asynchronous execution completes.

Return value

Type: [HRESULT](#)

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

Requirements

Header	D3DX11async.h
Library	D3DX11.lib

See also

[D3DX Functions](#)

D3DX11SaveTextureToFile function

11/2/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

NOTE

Instead of using this function, we recommend that you use the [DirectXTex](#) library, [CaptureTexture](#) then [SaveToXXXFile](#) (where XXX is WIC, DDS, or TGA; WIC doesn't support DDS and TGA; D3DX 9 supported TGA as a common art source format for games). For the simplified scenario of creating a screen shot from a render target texture, we recommend that you use the [DirectXTK](#) library, [SaveDDSTextureToFile](#) or [SaveWICTextureToFile](#).

Save a texture to a file.

Syntax

```
HRESULT D3DX11SaveTextureToFile(
    ID3D11DeviceContext      *pContext,
    _In_ ID3D11Resource       *pSrcTexture,
    _In_ D3DX11_IMAGE_FILE_FORMAT DestFormat,
    _In_ LPCTSTR              pDestFile
);
```

Parameters

pContext

Type: [ID3D11DeviceContext*](#)

A pointer to an [ID3D11DeviceContext](#) object.

pSrcTexture [in]

Type: [ID3D11Resource*](#)

Pointer to the texture to be saved. See [ID3D11Resource](#).

DestFormat [in]

Type: [D3DX11_IMAGE_FILE_FORMAT](#)

The format the texture will be saved as (see [D3DX11_IMAGE_FILE_FORMAT](#)). D3DX11_IFF_DDS is the preferred format since it is the only option that supports all the formats in [DXGI_FORMAT](#).

pDestFile [in]

Type: [LPCTSTR](#)

Name of the destination output file where the texture will be saved. If the compiler settings require Unicode, the

data type LPCTSTR resolves to LPCWSTR. Otherwise, the data type resolves to LPCSTR.

Return value

Type: [HRESULT](#)

The return value is one of the values listed in [Direct3D 11 Return Codes](#); use the return value to see if the *DestFormat* is supported.

Remarks

D3DX11SaveTextureToFile writes out the extra [DDS_HEADER_DXT10](#) structure for the input texture only if necessary (for example, because the input texture is in standard RGB (sRGB) format). If **D3DX11SaveTextureToFile** writes out the [DDS_HEADER_DXT10](#) structure, it sets the **dwFourCC** member of the [DDS_PIXELFORMAT](#) structure for the texture to **DX10** to indicate the presence of the [DDS_HEADER_DXT10](#) extended header.

Requirements

Header	D3DX11tex.h
Library	D3DX11.lib

See also

[D3DX Functions](#)

D3DX11SaveTextureToMemory function

11/2/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

NOTE

Instead of using this function, we recommend that you use the [DirectXTex](#) library, [CaptureTexture](#) then [SaveToXXXMemory](#) (where XXX is WIC, DDS, or TGA; WIC doesn't support DDS and TGA; D3DX 9 supported TGA as a common art source format for games).

Save a texture to memory.

Syntax

```
HRESULT D3DX11SaveTextureToMemory(
    ID3D11DeviceContext      *pContext,
    _In_   ID3D11Resource     *pSrcTexture,
    _In_   D3DX11_IMAGE_FILE_FORMAT DestFormat,
    _Out_  LPD3D10BLOB        *ppDestBuf,
    _In_   UINT                Flags
);
```

Parameters

pContext

Type: [ID3D11DeviceContext*](#)

A pointer to an [ID3D11DeviceContext](#) object.

pSrcTexture [in]

Type: [ID3D11Resource*](#)

Pointer to the texture to be saved. See [ID3D11Resource](#).

DestFormat [in]

Type: [D3DX11_IMAGE_FILE_FORMAT](#)

The format the texture will be saved as. See [D3DX11_IMAGE_FILE_FORMAT](#).

ppDestBuf [out]

Type: [LPD3D10BLOB*](#)

Address of a pointer to the memory containing the saved texture.

Flags [in]

Type: [UINT](#)

Optional.

Return value

Type: [HRESULT](#)

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

Requirements

Header	D3DX11tex.h
Library	D3DX11.lib

See also

[D3DX Functions](#)

D3DX11SHProjectCubeMap function

11/2/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

NOTE

Instead of using this function, we recommend that you use the [Spherical Harmonics Math](#) library, [SHProjectCubeMap](#).

Projects a function represented in a cube map into spherical harmonics.

Syntax

```
HRESULT D3DX11SHProjectCubeMap(
    ID3D11DeviceContext *pContext,
    UINT                Order,
    ID3D11Texture2D    *pCubeMap,
    FLOAT               *pROut,
    FLOAT               *pGOut,
    FLOAT               *pBOut
);
```

Parameters

pContext

Type: [ID3D11DeviceContext*](#)

A pointer to an [ID3D11DeviceContext](#) object.

Order

Type: [UINT](#)

Order of the SH evaluation, generates Order^2 coefficients whose degree is $\text{Order}-1$. Valid range is between 2 and 6.

pCubeMap

Type: [ID3D11Texture2D*](#)

A pointer to an [ID3D11Texture2D](#) that represents a cubemap that is going to be projected into spherical harmonics.

pROut

Type: [FLOAT*](#)

Output SH vector for red.

pGOut

Type: **FLOAT***

Output SH vector for green.

pBOut

Type: **FLOAT***

Output SH vector for blue.

Return value

Type: **HRESULT**

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

Requirements

Header	D3DX11tex.h
Library	D3DX11.lib

See also

[D3DX Functions](#)

D3DX11UnsetAllDeviceObjects function

2/22/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

NOTE

Instead of using this function, we recommend that you use the [ID3D11DeviceContext::ClearState](#) method.

Removes all resources from the device by setting their pointers to **NULL**. This should be called during shutdown of your application. It helps ensure that when one is releasing all of their resources that none of them are bound to the device.

Syntax

```
HRESULT D3DX11UnsetAllDeviceObjects(  
    _In_ ID3D11DeviceContext *pContext  
>;
```

Parameters

pContext [in]

Type: [ID3D11DeviceContext*](#)

Pointer to an [ID3D11DeviceContext](#) object.

Return value

Type: [HRESULT](#)

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

Requirements

Header	D3DX11core.h
Library	D3DX11.lib

See also

D3DX Functions

This section lists the functions provided by the D3DX library.

The functions are categorized into the following groups:

- Matrix Operations

- Vector Operations

- Color Operations

- Transform Operations

- Image Processing

- Font Operations

- Lighting Operations

- Geometry Operations

- Animation Operations

- Physics Operations

- Utility Functions

- File I/O Functions

- Memory Management Functions

- Thread Management Functions

- CriticalSection Management Functions

- Event Management Functions

- Semaphore Management Functions

- Memory Pool Management Functions

- Hash Table Management Functions

- Binary Tree Management Functions

- Priority Queue Management Functions

- Hash Map Management Functions

- Graph Management Functions

- Knapsack Problem Solving Functions

- Traveling Salesman Problem Solving Functions

- Job Scheduling Problem Solving Functions

- Resource Allocation Problem Solving Functions

- Knapsack Problem Solving Functions

- Traveling Salesman Problem Solving Functions

- Job Scheduling Problem Solving Functions

- Resource Allocation Problem Solving Functions

- Knapsack Problem Solving Functions

- Traveling Salesman Problem Solving Functions

- Job Scheduling Problem Solving Functions

- Resource Allocation Problem Solving Functions

- Knapsack Problem Solving Functions

- Traveling Salesman Problem Solving Functions

- Job Scheduling Problem Solving Functions

- Resource Allocation Problem Solving Functions

- Knapsack Problem Solving Functions

- Traveling Salesman Problem Solving Functions

- Job Scheduling Problem Solving Functions

- Resource Allocation Problem Solving Functions

- Knapsack Problem Solving Functions

- Traveling Salesman Problem Solving Functions

- Job Scheduling Problem Solving Functions

- Resource Allocation Problem Solving Functions

- Knapsack Problem Solving Functions

- Traveling Salesman Problem Solving Functions

- Job Scheduling Problem Solving Functions

- Resource Allocation Problem Solving Functions

D3DX Structures (Direct3D 11 Graphics)

2/4/2021 • 2 minutes to read • [Edit Online](#)

This section contains information about the D3DX 11 structures.

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

In this section

TOPIC	DESCRIPTION
D3DX11_IMAGE_INFO	<p>[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p>Optionally provide information to texture loader APIs to control how textures get loaded. A value of D3DX11_DEFAULT for any of these parameters will cause D3DX to automatically use the value from the source file.</p>
D3DX11_IMAGE_LOAD_INFO	<p>[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p>Optionally provide information to texture loader APIs to control how textures get loaded. A value of D3DX11_DEFAULT for any of these parameters will cause D3DX to automatically use the value from the source file.</p>
D3DX11_TEXTURE_LOAD_INFO	<p>[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p>Describes parameters used to load a texture from another texture.</p>

Related topics

[D3DX 11 Reference](#)

D3DX11_IMAGE_INFO structure

11/2/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

Optionally provide information to texture loader APIs to control how textures get loaded. A value of D3DX11_DEFAULT for any of these parameters will cause D3DX to automatically use the value from the source file.

Syntax

```
typedef struct D3DX11_IMAGE_INFO {
    UINT             Width;
    UINT             Height;
    UINT             Depth;
    UINT             ArraySize;
    UINT             MipLevels;
    UINT             MiscFlags;
    DXGI_FORMAT      Format;
    D3D11_RESOURCE_DIMENSION ResourceDimension;
    D3DX11_IMAGE_FILE_FORMAT ImageFileFormat;
} D3DX11_IMAGE_INFO, *LPD3DX11_IMAGE_INFO;
```

Members

Width

Type: [UINT](#)

The target width of the texture. If the actual width of the texture is larger or smaller than this value then the texture will be scaled up or down to fit this target width.

Height

Type: [UINT](#)

The target height of the texture. If the actual height of the texture is larger or smaller than this value then the texture will be scaled up or down to fit this target height.

Depth

Type: [UINT](#)

The depth of the texture. This only applies to volume textures.

ArraySize

Type: [UINT](#)

The number of elements in the array.

MipLevels

Type: [UINT](#)

The maximum number of mipmap levels in the texture. See the remarks in [D3D11_TEX1D_SRV](#). Using 0 or D3DX11_DEFAULT will cause a full mipmap chain to be created.

MiscFlags

Type: [UINT](#)

Miscellaneous resource properties specified with a [D3D11_RESOURCE_MISC_FLAG](#) flag.

Format

Type: [DXGI_FORMAT](#)

A [DXGI_FORMAT](#) enumeration specifying the format the texture will be in after it is loaded.

ResourceDimension

Type: [D3D11_RESOURCE_DIMENSION](#)

A [D3D11_RESOURCE_DIMENSION](#) value, which identifies the type of resource.

ImageFileFormat

Type: [D3DX11_IMAGE_FILE_FORMAT](#)

A [D3DX11_IMAGE_FILE_FORMAT](#) value, which identifies the image format.

Remarks

This structure is used by methods such as: [D3DX11GetImageInfoFromFile](#), [D3DX11GetImageInfoFromMemory](#), or [D3DX11GetImageInfoFromResource](#).

Requirements

Header	D3DX11tex.h
--------	-------------

See also

[D3DX Structures](#)

D3DX11_IMAGE_LOAD_INFO structure

11/2/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

Optionally provide information to texture loader APIs to control how textures get loaded. A value of D3DX11_DEFAULT for any of these parameters will cause D3DX to automatically use the value from the source file.

Syntax

```
typedef struct D3DX11_IMAGE_LOAD_INFO {
    UINT          Width;
    UINT          Height;
    UINT          Depth;
    UINT          FirstMipLevel;
    UINT          MipLevels;
    D3D11_USAGE   Usage;
    UINT          BindFlags;
    UINT          CpuAccessFlags;
    UINT          MiscFlags;
    DXGI_FORMAT   Format;
    UINT          Filter;
    UINT          MipFilter;
    D3DX11_IMAGE_INFO *pSrcInfo;
} D3DX11_IMAGE_LOAD_INFO, *LPD3DX11_IMAGE_LOAD_INFO;
```

Members

Width

Type: [UINT](#)

The target width of the texture. If the actual width of the texture is larger or smaller than this value then the texture will be scaled up or down to fit this target width.

Height

Type: [UINT](#)

The target height of the texture. If the actual height of the texture is larger or smaller than this value then the texture will be scaled up or down to fit this target height.

Depth

Type: [UINT](#)

The depth of the texture. This only applies to volume textures.

FirstMipLevel

Type: [UINT](#)

The highest resolution mipmap level of the texture. If this is greater than 0, then after the texture is loaded FirstMipLevel will be mapped to mipmap level 0.

MipLevels

Type: [UINT](#)

The maximum number of mipmap levels in the texture. See the remarks in [D3D11_TEX1D_SRV](#). Using 0 or D3DX11_DEFAULT will cause a full mipmap chain to be created.

Usage

Type: [D3D11_USAGE](#)

The way the texture resource is intended to be used. See [D3D11_USAGE](#).

BindFlags

Type: [UINT](#)

The pipeline stages that the texture will be allowed to bind to. See [D3D11_BIND_FLAG](#).

CpuAccessFlags

Type: [UINT](#)

The access permissions the cpu will have for the texture resource. See [D3D11_CPU_ACCESS_FLAG](#).

MiscFlags

Type: [UINT](#)

Miscellaneous resource properties (see [D3D11_RESOURCE_MISC_FLAG](#)).

Format

Type: [DXGI_FORMAT](#)

A [DXGI_FORMAT](#) enumeration indicating the format the texture will be in after it is loaded.

Filter

Type: [UINT](#)

Filter the texture using the specified filter (only when resampling). See [D3DX11_FILTER_FLAG](#).

MipFilter

Type: [UINT](#)

Filter the texture mip levels using the specified filter (only if generating mipmaps). Valid values are D3DX11_FILTER_NONE, D3DX11_FILTER_POINT, D3DX11_FILTER_LINEAR, or D3DX11_FILTER_TRIANGLE. See [D3DX11_FILTER_FLAG](#).

pSrcInfo

Type: [D3DX11_IMAGE_INFO](#)*

Information about the original image. See [D3DX11_IMAGE_INFO](#). Can be obtained with [D3DX11GetImageInfoFromFile](#), [D3DX11GetImageInfoFromMemory](#), or [D3DX11GetImageInfoFromResource](#).

Remarks

When initializing the structure, you may set any member to D3DX11_DEFAULT and D3DX will initialize it with a default value from the source texture when the texture is loaded.

This structure can be used by APIs that:

- Create resources, such as [D3DX11CreateTextureFromFile](#) and [D3DX11CreateShaderResourceViewFromFile](#).
- Create data processors, such as [D3DX11CreateAsyncTextureInfoProcessor](#) or [D3DX11CreateAsyncShaderResourceViewProcessor](#).

The default values are:

```
Width = D3DX11_DEFAULT;
Height = D3DX11_DEFAULT;
Depth = D3DX11_DEFAULT;
FirstMipLevel = D3DX11_DEFAULT;
MipLevels = D3DX11_DEFAULT;
Usage = (D3D11_USAGE) D3DX11_DEFAULT;
BindFlags = D3DX11_DEFAULT;
CpuAccessFlags = D3DX11_DEFAULT;
MiscFlags = D3DX11_DEFAULT;
Format = DXGI_FORMAT_FROM_FILE;
Filter = D3DX11_DEFAULT;
MipFilter = D3DX11_DEFAULT;
pSrcInfo = NULL;
```

Here is a brief example that uses this structure to supply the pixel format when loading a texture. For the complete code, see [HDR Tone Mapping CS11 Sample](#).

```
ID3D11ShaderResourceView* pCubeRV = NULL;
WCHAR strPath[MAX_PATH];
D3DX11_IMAGE_LOAD_INFO LoadInfo;

DXUTFindDXSDKMediaFileCch( strPath, MAX_PATH,
    L"Light Probes\\uffizi_cross.dds" );

LoadInfo.Format = DXGI_FORMAT_R16G16B16A16_FLOAT;

hr = D3DX11CreateShaderResourceViewFromFile( pd3dDevice, strPath,
    &LoadInfo, NULL, &pCubeRV, NULL );
```

Requirements

Header

D3DX11tex.h

See also

[D3DX Structures](#)

D3DX11_TEXTURE_LOAD_INFO structure

11/2/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

Describes parameters used to load a texture from another texture.

Syntax

```
typedef struct _D3DX11_TEXTURE_LOAD_INFO {
    D3D11_BOX *pSrcBox;
    D3D11_BOX *pDstBox;
    UINT        SrcFirstMip;
    UINT        DstFirstMip;
    UINT        NumMips;
    UINT        SrcFirstElement;
    UINT        DstFirstElement;
    UINT        NumElements;
    UINT        Filter;
    UINT        MipFilter;
} D3DX11_TEXTURE_LOAD_INFO;
```

Members

pSrcBox

Type: [D3D11_BOX*](#)

Source texture box (see [D3D11_BOX](#)).

pDstBox

Type: [D3D11_BOX*](#)

Destination texture box (see [D3D11_BOX](#)).

SrcFirstMip

Type: [UINT](#)

Source texture mipmap level, see [D3D11CalcSubresource](#) for more detail.

DstFirstMip

Type: [UINT](#)

Destination texture mipmap level, see [D3D11CalcSubresource](#) for more detail.

NumMips

Type: [UINT](#)

Number of mipmap levels in the source texture.

SrcFirstElement

Type: **UINT**

First element of the source texture.

DstFirstElement

Type: **UINT**

First element of the destination texture.

NumElements

Type: **UINT**

Number of elements to load.

Filter

Type: **UINT**

Filtering options during resampling (see [D3DX11_FILTER_FLAG](#)).

MipFilter

Type: **UINT**

Filtering options when generating mip levels (see [D3DX11_FILTER_FLAG](#)).

Remarks

This structure is used in a call to [D3DX11LoadTextureFromTexture](#).

The default values are:

```
pSrcBox = NULL;  
pDstBox = NULL;  
SrcFirstMip = 0;  
DstFirstMip = 0;  
NumMips = D3DX11_DEFAULT;  
SrcFirstElement = 0;  
DstFirstElement = 0;  
NumElements = D3DX11_DEFAULT;  
Filter = D3DX11_DEFAULT;  
MipFilter = D3DX11_DEFAULT;
```

Requirements

Header	D3DX11tex.h
--------	-------------

See also

[D3DX Structures](#)

D3DX Enumerations (Direct3D 11 Graphics)

2/4/2021 • 2 minutes to read • [Edit Online](#)

This section contains information about D3DX 11 enumerations.

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

In this section

TOPIC	DESCRIPTION
D3DX11_CHANNEL_FLAG	<p>[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p>These flags are used by functions which operate on one or more channels in a texture.</p>
D3DX11_ERR	<p>[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p>Errors are represented by negative values and cannot be combined. The following is a list of values that can be returned by methods included with the D3DX utility library. See the individual method descriptions for lists of the values that each can return. These lists are not necessarily comprehensive.</p>
D3DX11_FILTER_FLAG	<p>[!Note] The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p>Texture filtering flags.</p>

Topic	Description
D3DX11_IMAGE_FILE_FORMAT	<p data-bbox="842 197 922 226">[!Note]</p> <p data-bbox="842 231 1350 323">The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p data-bbox="822 406 1369 467">Image file formats supported by D3DX11CreateXXX and D3DX11SaveXXX functions.</p>
D3DX11_NORMALMAP_FLAG	<p data-bbox="842 557 922 586">[!Note]</p> <p data-bbox="842 590 1350 682">The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p data-bbox="822 759 1417 819">Normal map options. You can combine any number of these flags by using a bitwise OR operation.</p>
D3DX11_SAVE_TEXTURE_FLAG	<p data-bbox="842 902 922 932">[!Note]</p> <p data-bbox="842 936 1350 1028">The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.</p> <p data-bbox="822 1105 1033 1134">Texture save options.</p>

Related topics

[D3DX 11 Reference](#)

D3DX11_CHANNEL_FLAG enumeration

2/22/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

These flags are used by functions which operate on one or more channels in a texture.

Syntax

```
typedef enum D3DX11_CHANNEL_FLAG {
    D3DX11_CHANNEL_RED      = (1 << 0),
    D3DX11_CHANNEL_BLUE     = (1 << 1),
    D3DX11_CHANNEL_GREEN    = (1 << 2),
    D3DX11_CHANNEL_ALPHA    = (1 << 3),
    D3DX11_CHANNEL_LUMINANCE = (1 << 4)
} D3DX11_CHANNEL_FLAG, *LPD3DX11_CHANNEL_FLAG;
```

Constants

D3DX11_CHANNEL_RED

Indicates the red channel should be used.

D3DX11_CHANNEL_BLUE

Indicates the blue channel should be used.

D3DX11_CHANNEL_GREEN

Indicates the green channel should be used.

D3DX11_CHANNEL_ALPHA

Indicates the alpha channel should be used.

D3DX11_CHANNEL_LUMINANCE

Indicates the luminances of the red, green, and blue channels should be used.

Requirements

Header	D3DX11tex.h
--------	-------------

See also

[D3DX Enumerations](#)

D3DX11_ERR enumeration

2/22/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

Errors are represented by negative values and cannot be combined. The following is a list of values that can be returned by methods included with the D3DX utility library. See the individual method descriptions for lists of the values that each can return. These lists are not necessarily comprehensive.

Syntax

```
typedef enum D3DX11_ERR {  
    D3DX11_ERR_CANNOT_MODIFY_INDEX_BUFFER = MAKE_DDHRESULT(2900),  
    D3DX11_ERR_INVALID_MESH = MAKE_DDHRESULT(2901),  
    D3DX11_ERR_CANNOT_ATTR_SORT = MAKE_DDHRESULT(2902),  
    D3DX11_ERR_SKINNING_NOT_SUPPORTED = MAKE_DDHRESULT(2903),  
    D3DX11_ERR_TOO_MANY_INFLUENCES = MAKE_DDHRESULT(2904),  
    D3DX11_ERR_INVALID_DATA = MAKE_DDHRESULT(2905),  
    D3DX11_ERR_LOADED_MESH_HAS_NO_DATA = MAKE_DDHRESULT(2906),  
    D3DX11_ERR_DUPLICATE_NAMED_FRAGMENT = MAKE_DDHRESULT(2907),  
    D3DX11_ERR_CANNOT_REMOVE_LAST_ITEM = MAKE_DDHRESULT(2908)  
} D3DX11_ERR, *LPD3DX11_ERR;
```

Constants

D3DX11_ERR_CANNOT_MODIFY_INDEX_BUFFER

The index buffer cannot be modified.

D3DX11_ERR_INVALID_MESH

The mesh is invalid.

D3DX11_ERR_CANNOT_ATTR_SORT

Attribute sort (D3DXMESHOPT_ATTRSORT) is not supported as an optimization technique.

D3DX11_ERR_SKINNING_NOT_SUPPORTED

Skinning is not supported.

D3DX11_ERR_TOO_MANY_INFLUENCES

Too many influences specified.

D3DX11_ERR_INVALID_DATA

The data is invalid.

D3DX11_ERR_LOADED_MESH_HAS_NO_DATA

The mesh has no data.

D3DX11_ERR_DUPLICATE_NAMED_FRAGMENT

A fragment with that name already exists.

D3DX11_ERR_CANNOT_REMOVE_LAST_ITEM

The last item cannot be deleted.

Remarks

The facility code _FACDD is used to generate error codes, as in the following macros.

```
#define _FACDD          0x876
#define MAKE_DDHRESULT( code )  MAKE_HRESULT( 1, _FACDD, code )
enum _D3DXERR {
    D3DXERR_CANNOTMODIFYINDEXBUFFER = MAKE_DDHRESULT(2900),
    D3DXERR_INVALIDMESH           = MAKE_DDHRESULT(2901),
    ...
};
```

Requirements

Header

D3DX11.h

See also

[D3DX Enumerations](#)

D3DX11_FILTER_FLAG enumeration

2/22/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

Texture filtering flags.

Syntax

```
typedef enum D3DX11_FILTER_FLAG {
    D3DX11_FILTER_NONE          = (1 << 0),
    D3DX11_FILTER_POINT         = (2 << 0),
    D3DX11_FILTER_LINEAR        = (3 << 0),
    D3DX11_FILTER_TRIANGLE      = (4 << 0),
    D3DX11_FILTER_BOX           = (5 << 0),
    D3DX11_FILTER_MIRROR_U      = (1 << 16),
    D3DX11_FILTER_MIRROR_V      = (2 << 16),
    D3DX11_FILTER_MIRROR_W      = (4 << 16),
    D3DX11_FILTER_MIRROR        = (7 << 16),
    D3DX11_FILTER_DITHER         = (1 << 19),
    D3DX11_FILTER_DITHER_DIFFUSION = (2 << 19),
    D3DX11_FILTER_SRGB_IN        = (1 << 21),
    D3DX11_FILTER_SRGB_OUT       = (2 << 21),
    D3DX11_FILTER_SRGB           = (3 << 21)
} D3DX11_FILTER_FLAG, *LPD3DX11_FILTER_FLAG;
```

Constants

D3DX11_FILTER_NONE

No scaling or filtering will take place. Pixels outside the bounds of the source image are assumed to be transparent black.

D3DX11_FILTER_POINT

Each destination pixel is computed by sampling the nearest pixel from the source image.

D3DX11_FILTER_LINEAR

Each destination pixel is computed by sampling the four nearest pixels from the source image. This filter works best when the scale on both axes is less than two.

D3DX11_FILTER_TRIANGLE

Every pixel in the source image contributes equally to the destination image. This is the slowest of the filters.

D3DX11_FILTER_BOX

Each pixel is computed by averaging a 2x2(x2) box of pixels from the source image. This filter works only when the dimensions of the destination are half those of the source, as is the case with mipmaps.

D3DX11_FILTER_MIRROR_U

Pixels off the edge of the texture on the u-axis should be mirrored, not wrapped.

D3DX11_FILTER_MIRROR_V

Pixels off the edge of the texture on the v-axis should be mirrored, not wrapped.

D3DX11_FILTER_MIRROR_W

Pixels off the edge of the texture on the w-axis should be mirrored, not wrapped.

D3DX11_FILTER_MIRROR

Specifying this flag is the same as specifying the D3DX_FILTER_MIRROR_U, D3DX_FILTER_MIRROR_V, and D3DX_FILTER_MIRROR_W flags.

D3DX11_FILTER_DITHER

The resulting image must be dithered using a 4x4 ordered dither algorithm. This happens when converting from one format to another.

D3DX11_FILTER_DITHER_DIFFUSION

Do diffuse dithering on the image when changing from one format to another.

D3DX11_FILTER_SRGB_IN

Input data is in standard RGB (sRGB) color space. See remarks.

D3DX11_FILTER_SRGB_OUT

Output data is in standard RGB (sRGB) color space. See remarks.

D3DX11_FILTER_SRGB

Same as specifying D3DX_FILTER_SRGB_IN | D3DX_FILTER_SRGB_OUT. See remarks.

Remarks

D3DX11 automatically performs gamma correction (to convert color data from RGB space to standard RGB space) when loading texture data. This is automatically done for instance when RGB data is loaded from a .png file into an sRGB texture. Use the SRGB filter flags to indicate if the data does not need to be converted into sRGB space.

Requirements

Header	D3DX11tex.h
--------	-------------

See also

[D3DX Enumerations](#)

D3DX11_IMAGE_FILE_FORMAT enumeration

11/2/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

Image file formats supported by D3DX11CreateXXX and D3DX11SaveXXX functions.

Syntax

```
typedef enum D3DX11_IMAGE_FILE_FORMAT {
    D3DX11_IFF_BMP      = 0,
    D3DX11_IFF_JPG      = 1,
    D3DX11_IFF_PNG      = 3,
    D3DX11_IFF_DDS      = 4,
    D3DX11_IFF_TIFF     = 10,
    D3DX11_IFF_GIF      = 11,
    D3DX11_IFF_WMP      = 12,
    D3DX11_IFF_FORCE_DWORD = 0x7fffffff
} D3DX11_IMAGE_FILE_FORMAT, *LPD3DX11_IMAGE_FILE_FORMAT;
```

Constants

D3DX11_IFF_BMP

Windows bitmap (BMP) file format. Contains a header that describes the resolution of the device on which the rectangle of pixels was created, the dimensions of the rectangle, the size of the array of bits, a logical palette, and an array of bits that defines the relationship between pixels in the bitmapped image and entries in the logical palette. The file extension for this format is .bmp.

D3DX11_IFF_JPG

Joint Photographic Experts Group (JPEG) compressed file format. Specifies variable compression of 24-bit RGB color and 8-bit gray-scale Tagged Image File Format (TIFF) image document files. The file extension for this format is .jpg.

D3DX11_IFF_PNG

Portable Network Graphics (PNG) file format. A non-proprietary bitmap format using lossless compression. The file extension for this format is .png.

D3DX11_IFF_DDS

DirectDraw surface (DDS) file format. Stores textures, volume textures, and cubic environment maps, with or without mipmap levels, and with or without pixel compression. The file extension for this format is .dds.

D3DX11_IFF_TIFF

Tagged Image File Format (TIFF). The file extensions for this format are .tif and .tiff.

D3DX11_IFF_GIF

Graphics Interchange Format (GIF). The file extension for this format is .gif.

D3DX11_IFF_WMP

Windows Media Photo format (WMP). This format is also known as HD Photo and JPEG XR. The file extensions for this format are .hdp, .jxr, and .wdp.

To work properly, **D3DX11_IFF_WMP** requires that you initialize COM. Therefore, call [CoInitialize](#) or [CoInitializeEx](#) in your application before you call D3DX.

D3DX11_IFF_FORCE_DWORD

Forces this enumeration to compile to 32 bits in size. Without this value, some compilers would allow this enumeration to compile to a size other than 32 bits. This value is not used.

Remarks

See [Types of Bitmaps \(GDI+\)](#) for more information on some of these formats.

Requirements

Header	D3DX11tex.h
--------	-------------

See also

[D3DX Enumerations](#)

D3DX11_NORMALMAP_FLAG enumeration

11/2/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

Normal map options. You can combine any number of these flags by using a bitwise OR operation.

Syntax

```
typedef enum D3DX11_NORMALMAP_FLAG {  
    D3DX11_NORMALMAP_MIRROR_U          = (1 << 16),  
    D3DX11_NORMALMAP_MIRROR_V          = (2 << 16),  
    D3DX11_NORMALMAP_MIRROR           = (3 << 16),  
    D3DX11_NORMALMAP_INVERTSIGN       = (8 << 16),  
    D3DX11_NORMALMAP_COMPUTE_OCCLUSION = (16 << 16)  
} D3DX11_NORMALMAP_FLAG, *LPD3DX11_NORMALMAP_FLAG;
```

Constants

D3DX11_NORMALMAP_MIRROR_U

Indicates that pixels off the edge of the texture on the U-axis should be mirrored, not wrapped.

D3DX11_NORMALMAP_MIRROR_V

Indicates that pixels off the edge of the texture on the V-axis should be mirrored, not wrapped.

D3DX11_NORMALMAP_MIRROR

Same as D3DX11_NORMALMAP_MIRROR_U | D3DX11_NORMALMAP_MIRROR_V.

D3DX11_NORMALMAP_INVERTSIGN

Inverts the direction of each normal.

D3DX11_NORMALMAP_COMPUTE_OCCLUSION

Computes the per pixel occlusion term and encodes it into the alpha. An Alpha of 1 means that the pixel is not obscured in any way, and an alpha of 0 would mean that the pixel is completely obscured.

Remarks

These flags are used by [D3DX11ComputeNormalMap](#).

Requirements

Header

D3DX11tex.h

See also

[D3DX Enumerations](#)

D3DX11_SAVE_TEXTURE_FLAG enumeration

2/22/2020 • 2 minutes to read • [Edit Online](#)

NOTE

The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8 and is not supported for Windows Store apps.

Texture save options.

Syntax

```
typedef enum D3DX11_SAVE_TEXTURE_FLAG {  
    D3DX11_STF_USEINPUTBLOB = 0x0001  
} D3DX11_SAVE_TEXTURE_FLAG;
```

Constants

D3DX11_STF_USEINPUTBLOB

Do not optimize.

Requirements

Header	D3DX11tex.h
--------	-------------

See also

[D3DX Enumerations](#)

Effects 11 Reference

2/4/2021 • 2 minutes to read • [Edit Online](#)

Use Effects 11 source to build your effects-type application. The Effects 11 source is available at [Effects for Direct3D 11 Update](#). The Effects 11 API is described in this section.

In this section

TOPIC	DESCRIPTION
Effects 11 Interfaces	This section contains reference information for the component object model (COM) interfaces of the Effects 11 source.
Effects 11 Functions	This section contains information about the Effects 11 functions.
Effects 11 Structures	This section contains information about the Effects 11 structures.
Effect Format	An effect (which is often stored in a file with a .fx file extension) declares the pipeline state set by an effect. Effect state can be roughly broken down into three categories:

Related topics

[Direct3D 11 Reference](#)

Effects 11 Interfaces

2/4/2021 • 2 minutes to read • [Edit Online](#)

This section contains reference information for the component object model (COM) interfaces of the Effects 11 source.

In this section

TOPIC	DESCRIPTION
ID3DX11Effect	An ID3DX11Effect interface manages a set of state objects, resources, and shaders for implementing a rendering effect.
ID3DX11EffectBlendVariable	The blend-variable interface accesses blend state.
ID3DX11EffectClassInstanceVariable	Accesses a class instance.
ID3DX11EffectConstantBuffer	A constant-buffer interface accesses constant buffers or texture buffers.
ID3DX11EffectDepthStencilVariable	A depth-stencil-variable interface accesses depth-stencil state.
ID3DX11EffectDepthStencilViewVariable	A depth-stencil-view-variable interface accesses a depth-stencil view.
ID3DX11EffectGroup	The ID3DX11EffectGroup interface accesses an Effect group. The lifetime of an ID3DX11EffectGroup object is equal to the lifetime of its parent ID3DX11Effect object.
ID3DX11EffectInterfaceVariable	Accesses an interface variable.
ID3DX11EffectMatrixVariable	A matrix-variable interface accesses a matrix.
ID3DX11EffectPass	An ID3DX11EffectPass interface encapsulates state assignments within a technique. The lifetime of an ID3DX11EffectPass object is equal to the lifetime of its parent ID3DX11Effect object.
ID3DX11EffectRasterizerVariable	A rasterizer-variable interface accesses rasterizer state.
ID3DX11EffectRenderTargetViewVariable	A render-target-view interface accesses a render target.
ID3DX11EffectSamplerVariable	A sampler interface accesses sampler state.
ID3DX11EffectScalarVariable	An effect-scalar-variable interface accesses scalar values.
ID3DX11EffectShaderResourceVariable	A shader-resource interface accesses a shader resource.

TOPIC	DESCRIPTION
ID3DX11EffectShaderVariable	A shader-variable interface accesses a shader variable.
ID3DX11EffectStringVariable	A string-variable interface accesses a string variable.
ID3DX11EffectTechnique	<p>An ID3DX11EffectTechnique interface is a collection of passes. The lifetime of an ID3DX11EffectTechnique object is equal to the lifetime of its parent ID3DX11Effect object.</p>
ID3DX11EffectType	<p>The ID3DX11EffectType interface accesses effect variables by type. The lifetime of an ID3DX11EffectType object is equal to the lifetime of its parent ID3DX11Effect object.</p>
ID3DX11EffectUnorderedAccessViewVariable	Accesses an unordered access view.
ID3DX11EffectVariable	<p>The ID3DX11EffectVariable interface is the base class for all effect variables. The lifetime of an ID3DX11EffectVariable object is equal to the lifetime of its parent ID3DX11Effect object.</p>
ID3DX11EffectVectorVariable	A vector-variable interface accesses a four-component vector.

Related topics

[Effects 11 Reference](#)

ID3DX11Effect interface

11/2/2020 • 2 minutes to read • [Edit Online](#)

An **ID3DX11Effect** interface manages a set of state objects, resources, and shaders for implementing a rendering effect.

Members

The **ID3DX11Effect** interface inherits from the [IUnknown](#) interface. **ID3DX11Effect** also has these types of members:

- [Methods](#)

Methods

The **ID3DX11Effect** interface has these methods.

METHOD	DESCRIPTION
CloneEffect	Creates a copy of an effect interface.
GetClassLinkage	Gets a class linkage interface.
GetConstantBufferByIndex	Get a constant buffer by index.
GetConstantBufferByName	Get a constant buffer by name.
GetDesc	Get an effect description.
GetDevice	Get the device that created the effect.
GetGroupByIndex	Gets an effect group by index.
GetGroupByName	Gets an effect group by name.
GetTechniqueByIndex	Get a technique by index.
GetTechniqueByName	Get a technique by name.
GetVariableByIndex	Get a variable by index.
GetVariableByName	Get a variable by name.
GetVariableBySemantic	Get a variable by semantic.
IsOptimized	Test an effect to see if the reflection metadata has been removed from memory.
IsValid	Test an effect to see if it contains valid syntax.
Optimize	Minimize the amount of memory required for an effect.

Remarks

An effect is created by calling [D3DX11CreateEffectFromMemory](#).

The effect system groups the information required for rendering into an effect which contains: state objects for assigning state changes in groups, resources for supplying input data and storing output data, and programs that control how the rendering is done called shaders.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

NOTE

If you call [QueryInterface](#) on an [ID3DX11Effect](#) object to retrieve the [IUnknown](#) interface, [QueryInterface](#) returns [E_NOINTERFACE](#). To work around this issue, use the following code:

```
IUnknown* pIUnknown = (IUnknown*)pEffect;  
pIUnknown->AddRef();
```

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[Effects 11 Interfaces](#)

[D3DX Interfaces](#)

> > > > >

ID3DX11Effect::CloneEffect method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Creates a copy of an effect interface.

Syntax

```
HRESULT CloneEffect(  
    UINT          Flags,  
    ID3DX11Effect **ppClonedEffect  
>;
```

Parameters

Flags

Type: [UINT](#)

Flags affecting the creation of the cloned effect. Can be 0 or one of the following values.

FLAG	DESCRIPTION
D3DX11_EFFECT_CLONE_FORCE_NONSINGLE	Ignore all "single" qualifiers on cbuffers. All cbuffers will have their own ID3D11Buffer s created in the cloned effect.

ppClonedEffect

Type: [ID3DX11Effect**](#)

Pointer to an [ID3DX11Effect](#) pointer that will be set to the copy of the effect.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header

D3dx11effect.h

Library

N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11Effect](#)

ID3DX11Effect::GetClassLinkage method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Gets a class linkage interface.

Syntax

```
ID3D11ClassLinkage* GetClassLinkage();
```

Parameters

This method has no parameters.

Return value

Type: [ID3D11ClassLinkage*](#)

Returns a pointer to an [ID3D11ClassLinkage](#) interface.

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11Effect](#)

ID3DX11Effect::GetConstantBufferByIndex method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get a constant buffer by index.

Syntax

```
ID3DX11EffectConstantBuffer* GetConstantBufferByIndex(  
    UINT Index  
) ;
```

Parameters

Index

Type: [UINT](#)

A zero-based index.

Return value

Type: [ID3DX11EffectConstantBuffer*](#)

A pointer to a [ID3DX11EffectConstantBuffer](#).

Remarks

An effect that contains a variable that will be read/written by an application requires at least one constant buffer. For best performance, an effect should organize variables into one or more constant buffers based on their frequency of update.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

ID3DX11Effect

ID3DX11Effect::GetConstantBufferByName method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get a constant buffer by name.

Syntax

```
ID3DX11EffectConstantBuffer* GetConstantBufferByName(  
    LPCSTR Name  
) ;
```

Parameters

Name

Type: [LPCSTR](#)

The constant-buffer name.

Return value

Type: [ID3DX11EffectConstantBuffer*](#)

A pointer to the constant buffer indicated by the Name. See [ID3DX11EffectConstantBuffer](#).

Remarks

An effect that contains a variable that will be read/written by an application requires at least one constant buffer. For best performance, an effect should organize variables into one or more constant buffers based on their frequency of update.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

ID3DX11Effect

ID3DX11Effect::GetDesc method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get an effect description.

Syntax

```
HRESULT GetDesc(  
    D3DX11_EFFECT_DESC *pDesc  
>;
```

Parameters

pDesc

Type: [D3DX11_EFFECT_DESC*](#)

A pointer to an effect description (see [D3DX11_EFFECT_DESC](#)).

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

An effect description contains basic information about an effect such as the techniques it contains and the constant buffer resources it requires.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11Effect](#)

ID3DX11Effect::GetDevice method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get the device that created the effect.

Syntax

```
HRESULT GetDevice(  
    ID3D11Device **ppDevice  
>;
```

Parameters

ppDevice

Type: [ID3D11Device**](#)

A pointer to an [ID3D11Device](#).

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

An effect is created for a specific device, by calling a function such as [D3DX11CreateEffectFromMemory](#).

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11Effect](#)

ID3DX11Effect::GetGroupByIndex method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Gets an effect group by index.

Syntax

```
ID3DX11EffectGroup* GetGroupByIndex(  
    UINT Index  
) ;
```

Parameters

Index

Type: [UINT](#)

Index of the effect group.

Return value

Type: [ID3DX11EffectGroup*](#)

A pointer to an [ID3DX11EffectGroup](#) interface.

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11Effect](#)

ID3DX11Effect::GetGroupByName method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Gets an effect group by name.

Syntax

```
ID3DX11EffectGroup* GetGroupByName(  
    LPCSTR Name  
) ;
```

Parameters

Name

Type: [LPCSTR](#)

Name of the effect group.

Return value

Type: [ID3DX11EffectGroup*](#)

A pointer to an [ID3DX11EffectGroup](#) interface.

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11Effect](#)

ID3DX11Effect::GetTechniqueByIndex method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get a technique by index.

Syntax

```
ID3DX11EffectTechnique* GetTechniqueByIndex(  
    UINT Index  
) ;
```

Parameters

Index

Type: [UINT](#)

A zero-based index.

Return value

Type: [ID3DX11EffectTechnique*](#)

A pointer to an [ID3DX11EffectTechnique](#).

Remarks

An effect contains one or more techniques; each technique contains one or more passes. You can access a technique using its name or with an index.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11Effect](#)

ID3DX11Effect::GetTechniqueByName method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get a technique by name.

Syntax

```
ID3DX11EffectTechnique* GetTechniqueByName(  
    LPCSTR Name  
) ;
```

Parameters

Name

Type: [LPCSTR](#)

The name of the technique.

Return value

Type: [ID3DX11EffectTechnique*](#)

A pointer to an [ID3DX11EffectTechnique](#). If a technique with the appropriate name is not found an invalid technique is returned. [ID3DX11EffectTechnique::IsValid](#) should be called on the returned technique to determine whether it is valid.

Remarks

An effect contains one or more techniques; each technique contains one or more passes. You can access a technique using its name or with an index.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

ID3DX11Effect

ID3DX11Effect::GetVariableByIndex method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get a variable by index.

Syntax

```
ID3DX11EffectVariable* GetVariableByIndex(  
    UINT Index  
) ;
```

Parameters

Index

Type: [UINT](#)

A zero-based index.

Return value

Type: [ID3DX11EffectVariable*](#)

A pointer to a [ID3DX11EffectVariable](#).

Remarks

An effect may contain one or more variables. Variables outside of a technique are considered global to all effects, those located inside of a technique are local to that technique. You can access any local non-static effect variable using its name or with an index.

The method returns a pointer to an [effect-variable interface](#) if a variable is not found; you can call [ID3DX11Effect::IsValid](#) to verify whether or not the index exists.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11Effect](#)

ID3DX11Effect::GetVariableByName method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get a variable by name.

Syntax

```
ID3DX11EffectVariable* GetVariableByName(  
    LPCSTR Name  
)
```

Parameters

Name

Type: [LPCSTR](#)

The variable name.

Return value

Type: [ID3DX11EffectVariable*](#)

A pointer to an [ID3DX11EffectVariable](#). Returns an invalid variable if the specified name cannot be found.

Remarks

An effect may contain one or more variables. Variables outside of a technique are considered global to all effects, those located inside of a technique are local to that technique. You can access an effect variable using its name or with an index.

The method returns a pointer to an [effect-variable interface](#) whether or not a variable is found.

[ID3DX11Effect::IsValid](#) should be called to verify whether or not the name exists.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11Effect](#)

ID3DX11Effect::GetVariableBySemantic method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get a variable by semantic.

Syntax

```
ID3DX11EffectVariable* GetVariableBySemantic(  
    LPCSTR Semantic  
) ;
```

Parameters

Semantic

Type: [LPCSTR](#)

The semantic name.

Return value

Type: [ID3DX11EffectVariable*](#)

A pointer to the effect variable indicated by the Semantic. See [ID3DX11EffectVariable](#).

Remarks

Each effect variable can have a semantic attached, which is a user defined metadata string. Some [system-value semantics](#) are reserved words that trigger built in functionality by pipeline stages.

The method returns a pointer to an [effect-variable interface](#) if a variable is not found; you can call [ID3DX11Effect::IsValid](#) to verify whether or not the semantic exists.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11Effect](#)

ID3DX11Effect::IsOptimized method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Test an effect to see if the reflection metadata has been removed from memory.

Syntax

```
BOOL IsOptimized();
```

Parameters

This method has no parameters.

Return value

Type: **BOOL**

TRUE if the effect is optimized; otherwise FALSE.

Remarks

An effect uses memory space two different ways: to store the information required by the runtime to execute an effect, and to store the metadata required to reflect information back to an application using the API. You can minimize the amount of memory required by an effect by calling [ID3DX11Effect::Optimize](#) which removes the reflection metadata from memory. Of course, API methods to read variables will no longer work once reflection data has been removed.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11Effect](#)

ID3DX11Effect::IsValid method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Test an effect to see if it contains valid syntax.

Syntax

```
BOOL IsValid();
```

Parameters

This method has no parameters.

Return value

Type: **BOOL**

TRUE if the code syntax is valid; otherwise FALSE.

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11Effect](#)

ID3DX11Effect::Optimize method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Minimize the amount of memory required for an effect.

Syntax

```
HRESULT Optimize();
```

Parameters

This method has no parameters.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

An effect uses memory space two different ways: to store the information required by the runtime to execute an effect, and to store the metadata required to reflect information back to an application using the API. You can minimize the amount of memory required by an effect by calling **ID3DX11Effect::Optimize** which removes the reflection metadata from memory. API methods to read variables will no longer work once reflection data has been removed.

The following methods will fail after Optimize has been called on an effect.

- [ID3DX11Effect::GetConstantBufferByIndex](#)
- [ID3DX11Effect::GetConstantBufferByName](#)
- [ID3DX11Effect::GetDesc](#)
- [ID3DX11Effect::GetDevice](#)
- [ID3DX11Effect::GetTechniqueByIndex](#)
- [ID3DX11Effect::GetTechniqueByName](#)
- [ID3DX11Effect::GetVariableByIndex](#)
- [ID3DX11Effect::GetVariableByName](#)
- [ID3DX11Effect::GetVariableBySemantic](#)

NOTE

References retrieved with these methods before calling **ID3DX11Effect::Optimize** are still valid after **ID3DX11Effect::Optimize** is called. This allows the application to get all the variables, techniques, and passes it will use, call Optimize, and then use the effect.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11Effect](#)

ID3DX11EffectBlendVariable interface

2/22/2020 • 2 minutes to read • [Edit Online](#)

The blend-variable interface accesses blend state.

Members

The ID3DX11EffectBlendVariable interface inherits from [ID3DX11EffectVariable](#).

ID3DX11EffectBlendVariable also has these types of members:

- [Methods](#)

Methods

The ID3DX11EffectBlendVariable interface has these methods.

METHOD	DESCRIPTION
GetBackingStore	Get a pointer to a blend-state variable.
GetBlendState	Get a pointer to a blend-state interface.
SetBlendState	Sets an effect's blend-state.
UndoSetBlendState	Reverts a previously set blend-state.

Remarks

An [ID3DX11EffectVariable](#) interface is created when an effect is read into memory.

Effect variables are saved in memory in the backing store; when a technique is applied, the values in the backing store are copied to the device. You can use either of these methods to return state.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

ID3DX11EffectVariable

Effects 11 Interfaces

D3DX Interfaces

ID3DX11EffectBlendVariable::GetBackingStore method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get a pointer to a blend-state variable.

Syntax

```
HRESULT GetBackingStore(  
    UINT           Index,  
    D3D11_BLEND_DESC *pBlendDesc  
>;
```

Parameters

Index

Type: [UINT](#)

Index into an array of blend-state descriptions. If there is only one blend-state variable in the effect, use 0.

pBlendDesc

Type: [D3D11_BLEND_DESC*](#)

A pointer to a blend-state description (see [D3D11_BLEND_DESC](#)).

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

Effect variables are saved in memory in the backing store; when a technique is applied, the values in the backing store are copied to the device. Backing store data can be used to recreate the variable when necessary.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header

D3dx11effect.h

Library	N/A (An Effects 11 library is available online as shared source.)
---------	---

See also

[ID3DX11EffectBlendVariable](#)

ID3DX11EffectBlendVariable::GetBlendState method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get a pointer to a blend-state interface.

Syntax

```
HRESULT GetBlendState(  
    UINT             Index,  
    ID3D11BlendState **ppBlendState  
>;
```

Parameters

Index

Type: [UINT](#)

Index into an array of blend-state interfaces. If there is only one blend-state interface, use 0.

ppBlendState

Type: [ID3D11BlendState**](#)

The address of a pointer to a blend-state interface (see [ID3D11BlendState](#)).

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectBlendVariable](#)

ID3DX11EffectBlendVariable::SetBlendState method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Sets an effect's blend-state.

Syntax

```
HRESULT SetBlendState(  
    UINT           Index,  
    ID3D11BlendState *pBlendState  
>;
```

Parameters

Index

Type: **UINT**

Index into an array of blend-state interfaces. If there is only one blend-state interface, use 0.

pBlendState

Type: **ID3D11BlendState***

A pointer to an **ID3D11BlendState** interface containing the blend-state to set.

Return value

Type: **HRESULT**

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectBlendVariable](#)

ID3DX11EffectBlendVariable::UndoSetBlendState method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Reverts a previously set blend-state.

Syntax

```
HRESULT UndoSetBlendState(  
    UINT Index  
);
```

Parameters

Index

Type: **UINT**

Index into an array of blend-state interfaces. If there is only one blend-state interface, use 0.

Return value

Type: **HRESULT**

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectBlendVariable](#)

ID3DX11EffectClassInstanceVariable interface

2/22/2020 • 2 minutes to read • [Edit Online](#)

Accesses a class instance.

Members

The **ID3DX11EffectClassInstanceVariable** interface inherits from [ID3DX11EffectVariable](#).

ID3DX11EffectClassInstanceVariable also has these types of members:

- [Methods](#)

Methods

The **ID3DX11EffectClassInstanceVariable** interface has these methods.

METHOD	DESCRIPTION
GetClassInstance	Gets a class instance.

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVariable](#)

[Effects 11 Interfaces](#)

[D3DX Interfaces](#)

ID3DX11EffectClassInstanceVariable::GetClassInstance method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Gets a class instance.

Syntax

```
HRESULT GetClassInstance(  
    ID3D11ClassInstance **ppClassInstance  
>;
```

Parameters

ppClassInstance

Type: [ID3D11ClassInstance**](#)

Pointer to an [ID3D11ClassInstance](#) pointer that will be set to class instance.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectClassInstanceVariable](#)

ID3DX11EffectConstantBuffer interface

2/22/2020 • 2 minutes to read • [Edit Online](#)

A constant-buffer interface accesses constant buffers or texture buffers.

Members

The **ID3DX11EffectConstantBuffer** interface inherits from [ID3DX11EffectVariable](#).

ID3DX11EffectConstantBuffer also has these types of members:

- [Methods](#)

Methods

The **ID3DX11EffectConstantBuffer** interface has these methods.

METHOD	DESCRIPTION
GetConstantBuffer	Get a constant-buffer.
GetTextureBuffer	Get a texture-buffer.
SetConstantBuffer	Set a constant-buffer.
SetTextureBuffer	Set a texture-buffer.
UndoSetConstantBuffer	Reverts a previously set constant buffer.
UndoSetTextureBuffer	Reverts a previously set texture buffer.

Remarks

Use constant buffers to store many effect constants; grouping constants into buffers based on their frequency of update. This allows you to minimize the number of state changes as well as make the fewest API calls to change state. Both of these factors lead to better performance.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
--------	----------------

Library	N/A (An Effects 11 library is available online as shared source.)
---------	---

See also

[ID3DX11EffectVariable](#)

[Effects 11 Interfaces](#)

[D3DX Interfaces](#)

ID3DX11EffectConstantBuffer::GetConstantBuffer method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get a constant-buffer.

Syntax

```
HRESULT GetConstantBuffer(  
    ID3D11Buffer **ppConstantBuffer  
>;
```

Parameters

ppConstantBuffer

Type: [ID3D11Buffer**](#)

The address of a pointer to a constant-buffer interface. See [ID3D11Buffer](#).

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectConstantBuffer](#)

ID3DX11EffectConstantBuffer::GetTextureBuffer method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get a texture-buffer.

Syntax

```
HRESULT GetTextureBuffer(  
    ID3D11ShaderResourceView **ppTextureBuffer  
>;
```

Parameters

ppTextureBuffer

Type: [ID3D11ShaderResourceView**](#)

The address of a pointer to a shader-resource-view interface for accessing a texture buffer. See [ID3D11ShaderResourceView](#).

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectConstantBuffer](#)

ID3DX11EffectConstantBuffer::SetConstantBuffer method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Set a constant-buffer.

Syntax

```
HRESULT SetConstantBuffer(  
    ID3D11Buffer *pConstantBuffer  
>;
```

Parameters

pConstantBuffer

Type: [ID3D11Buffer*](#)

A pointer to a constant-buffer interface. See [ID3D11Buffer](#).

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectConstantBuffer](#)

ID3DX11EffectConstantBuffer::SetTextureBuffer method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Set a texture-buffer.

Syntax

```
HRESULT SetTextureBuffer(  
    ID3D11ShaderResourceView *pTextureBuffer  
>;
```

Parameters

pTextureBuffer

Type: [ID3D11ShaderResourceView*](#)

A pointer to a shader-resource-view interface for accessing a texture buffer.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectConstantBuffer](#)

ID3DX11EffectConstantBuffer::UndoSetConstantBuffer method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Reverts a previously set constant buffer.

Syntax

```
HRESULT UndoSetConstantBuffer();
```

Parameters

This method has no parameters.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectConstantBuffer](#)

ID3DX11EffectConstantBuffer::UndoSetTextureBuffer method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Reverts a previously set texture buffer.

Syntax

```
HRESULT UndoSetTextureBuffer();
```

Parameters

This method has no parameters.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectConstantBuffer](#)

ID3DX11EffectDepthStencilVariable interface

2/22/2020 • 2 minutes to read • [Edit Online](#)

A depth-stencil-variable interface accesses depth-stencil state.

Members

The ID3DX11EffectDepthStencilVariable interface inherits from [ID3DX11EffectVariable](#).

ID3DX11EffectDepthStencilVariable also has these types of members:

- [Methods](#)

Methods

The ID3DX11EffectDepthStencilVariable interface has these methods.

METHOD	DESCRIPTION
GetBackingStore	Get a pointer to a variable that contains depth-stencil state.
GetDepthStencilState	Get a pointer to a depth-stencil interface.
SetDepthStencilState	Sets the depth stencil state.
UndoSetDepthStencilState	Reverts a previously set depth stencil state.

Remarks

An [ID3DX11EffectVariable](#) interface is created when an effect is read into memory.

Effect variables are saved in memory in the backing store; when a technique is applied, the values in the backing store are copied to the device. You can use either of these methods to return state.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

ID3DX11EffectVariable

Effects 11 Interfaces

D3DX Interfaces

ID3DX11EffectDepthStencilVariable::GetBackingStore method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get a pointer to a variable that contains depth-stencil state.

Syntax

```
HRESULT GetBackingStore(  
    UINT             Index,  
    D3D11_DEPTH_STENCIL_DESC *pDepthStencilDesc  
)
```

Parameters

Index

Type: [UINT](#)

Index into an array of depth-stencil-state descriptions. If there is only one depth-stencil variable in the effect, use 0.

pDepthStencilDesc

Type: [D3D11_DEPTH_STENCIL_DESC*](#)

A pointer to a depth-stencil-state description (see [D3D11_DEPTH_STENCIL_DESC](#)).

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

Effect variables are saved in memory in the backing store; when a technique is applied, the values in the backing store are copied to the device. Backing store data can be used to recreate the variable when necessary.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header

D3dx11effect.h

Library	N/A (An Effects 11 library is available online as shared source.)
---------	---

See also

[ID3DX11EffectDepthStencilVariable](#)

ID3DX11EffectDepthStencilVariable::GetDepthStencilState method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get a pointer to a depth-stencil interface.

Syntax

```
HRESULT GetDepthStencilState(  
    UINT             Index,  
    ID3D11DepthStencilState **ppDepthStencilState  
) ;
```

Parameters

Index

Type: [UINT](#)

Index into an array of depth-stencil interfaces. If there is only one depth-stencil interface, use 0.

ppDepthStencilState

Type: [ID3D11DepthStencilState**](#)

The address of a pointer to a blend-state interface (see [ID3D11DepthStencilState](#)).

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectDepthStencilVariable](#)

ID3DX11EffectDepthStencilVariable::SetDepthStencilState method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Sets the depth stencil state.

Syntax

```
HRESULT SetDepthStencilState(  
    UINT             Index,  
    ID3D11DepthStencilState *pDepthStencilState  
) ;
```

Parameters

Index

Type: **UINT**

Index into an array of depth-stencil interfaces. If there is only one depth-stencil interface, use 0.

pDepthStencilState

Type: **ID3D11DepthStencilState***

Pointer to an **ID3D11DepthStencilState** interface containing the new depth stencil state.

Return value

Type: **HRESULT**

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectDepthStencilVariable](#)

ID3DX11EffectDepthStencilVariable::UndoSetDepthStencilState method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Reverts a previously set depth stencil state.

Syntax

```
HRESULT UndoSetDepthStencilState(  
    UINT Index  
)
```

Parameters

Index

Type: **UINT**

Index into an array of depth-stencil interfaces. If there is only one depth-stencil interface, use 0.

Return value

Type: **HRESULT**

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectDepthStencilVariable](#)

ID3DX11EffectDepthStencilViewVariable interface

2/22/2020 • 2 minutes to read • [Edit Online](#)

A depth-stencil-view-variable interface accesses a depth-stencil view.

Members

The ID3DX11EffectDepthStencilViewVariable interface inherits from [ID3DX11EffectVariable](#).

ID3DX11EffectDepthStencilViewVariable also has these types of members:

- [Methods](#)

Methods

The ID3DX11EffectDepthStencilViewVariable interface has these methods.

METHOD	DESCRIPTION
GetDepthStencil	Get a depth-stencil-view resource.
GetDepthStencilArray	Get an array of depth-stencil-view resources.
SetDepthStencil	Set a depth-stencil-view resource.
SetDepthStencilArray	Set an array of depth-stencil-view resources.

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVariable](#)

[Effects 11 Interfaces](#)

ID3DX11EffectDepthStencilViewVariable::GetDepthStencil method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get a depth-stencil-view resource.

Syntax

```
HRESULT GetDepthStencil(
    ID3D11DepthStencilView **ppResource
);
```

Parameters

ppResource

Type: [ID3D11DepthStencilView**](#)

The address of a pointer to a depth-stencil-view interface. See [ID3D11DepthStencilView](#).

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectDepthStencilViewVariable](#)

ID3DX11EffectDepthStencilViewVariable::GetDepthStencilArray method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get an array of depth-stencil-view resources.

Syntax

```
HRESULT GetDepthStencilArray(  
    ID3D11DepthStencilView **ppResources,  
    UINT             Offset,  
    UINT             Count  
>;
```

Parameters

ppResources

Type: [ID3D11DepthStencilView**](#)

A pointer to an array of depth-stencil-view interfaces. See [ID3D11DepthStencilView](#).

Offset

Type: [UINT](#)

The zero-based array index to get the first interface.

Count

Type: [UINT](#)

The number of elements in the array.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
--------	----------------

Library

N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectDepthStencilViewVariable](#)

ID3DX11EffectDepthStencilViewVariable::SetDepthStencil method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Set a depth-stencil-view resource.

Syntax

```
HRESULT SetDepthStencil(  
    ID3D11DepthStencilView *pResource  
>;
```

Parameters

pResource

Type: [ID3D11DepthStencilView*](#)

A pointer to a depth-stencil-view interface. See [ID3D11DepthStencilView](#).

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectDepthStencilViewVariable](#)

ID3DX11EffectDepthStencilViewVariable::SetDepthStencilArray method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Set an array of depth-stencil-view resources.

Syntax

```
HRESULT SetDepthStencilArray(  
    ID3D11DepthStencilView **ppResources,  
    UINT             Offset,  
    UINT             Count  
) ;
```

Parameters

ppResources

Type: [ID3D11DepthStencilView**](#)

A pointer to an array of depth-stencil-view interfaces. See [ID3D11DepthStencilView](#).

Offset

Type: [UINT](#)

The zero-based array index to set the first interface.

Count

Type: [UINT](#)

The number of elements in the array.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h

Library	N/A (An Effects 11 library is available online as shared source.)
---------	---

See also

[ID3DX11EffectDepthStencilViewVariable](#)

ID3DX11EffectGroup interface

2/22/2020 • 2 minutes to read • [Edit Online](#)

The **ID3DX11EffectGroup** interface accesses an Effect group.

The lifetime of an **ID3DX11EffectGroup** object is equal to the lifetime of its parent **ID3DX11Effect** object.

- [Methods](#)

Methods

The **ID3DX11EffectGroup** interface has these methods.

METHOD	DESCRIPTION
GetAnnotationByIndex	Get an annotation by index.
GetAnnotationByName	Get an annotation by name.
GetDesc	Gets a group description.
GetTechniqueByIndex	Get a technique by index.
GetTechniqueByName	Get a technique by name.
IsValid	Test an effect to see if it contains valid syntax.

Remarks

To get an **ID3DX11EffectGroup** interface, call a method like [ID3DX11Effect::GetGroupByName](#).

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[Effects 11 Interfaces](#)

ID3DX11EffectGroup::GetAnnotationByIndex method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get an annotation by index.

Syntax

```
ID3DX11EffectVariable* GetAnnotationByIndex(  
    UINT Index  
) ;
```

Parameters

Index

Type: [UINT](#)

Index of the annotation.

Return value

Type: [ID3DX11EffectVariable*](#)

Pointer to an [ID3DX11EffectVariable](#) interface.

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectGroup](#)

ID3DX11EffectGroup::GetAnnotationByName method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get an annotation by name.

Syntax

```
ID3DX11EffectVariable* GetAnnotationByName(  
    LPCSTR Name  
)
```

Parameters

Name

Type: [LPCSTR](#)

The name of the annotation.

Return value

Type: [ID3DX11EffectVariable*](#)

A pointer to an [ID3DX11EffectVariable](#). Note that if the annotation is not found the [ID3DX11EffectVariable](#) returned will be empty. The [ID3DX11EffectVariable::IsValid](#) method should be called to determine whether the annotation was found.

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

ID3DX11EffectGroup

ID3DX11EffectGroup::GetDesc method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Gets a group description.

Syntax

```
HRESULT GetDesc(  
    D3DX11_GROUP_DESC *pDesc  
)
```

Parameters

pDesc

Type: [D3DX11_GROUP_DESC*](#)

A pointer to a [D3DX11_GROUP_DESC](#) structure.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectGroup](#)

ID3DX11EffectGroup::GetTechniqueByIndex method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get a technique by index.

Syntax

```
ID3DX11EffectTechnique* GetTechniqueByIndex(  
    UINT Index  
) ;
```

Parameters

Index

Type: [UINT](#)

A zero-based index.

Return value

Type: [ID3DX11EffectTechnique*](#)

A pointer to an [ID3DX11EffectTechnique](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectGroup](#)

ID3DX11EffectGroup::GetTechniqueByName method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get a technique by name.

Syntax

```
ID3DX11EffectTechnique* GetTechniqueByName(  
    LPCSTR Name  
) ;
```

Parameters

Name

Type: [LPCSTR](#)

The name of the technique.

Return value

Type: [ID3DX11EffectTechnique*](#)

A pointer to an [ID3DX11EffectTechnique](#), or **NULL** if the technique is not found.

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectGroup](#)

ID3DX11EffectGroup::IsValid method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Test an effect to see if it contains valid syntax.

Syntax

```
BOOL IsValid();
```

Parameters

This method has no parameters.

Return value

Type: **BOOL**

TRUE if the code syntax is valid; otherwise FALSE.

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectGroup](#)

ID3DX11EffectInterfaceVariable interface

2/22/2020 • 2 minutes to read • [Edit Online](#)

Accesses an interface variable.

Members

The **ID3DX11EffectInterfaceVariable** interface inherits from [ID3DX11EffectVariable](#).

ID3DX11EffectInterfaceVariable also has these types of members:

- [Methods](#)

Methods

The **ID3DX11EffectInterfaceVariable** interface has these methods.

METHOD	DESCRIPTION
GetClassInstance	Get a class instance.
SetClassInstance	Sets a class instance.

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVariable](#)

[Effects 11 Interfaces](#)

[D3DX Interfaces](#)

ID3DX11EffectInterfaceVariable::GetClassInstance method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get a class instance.

Syntax

```
HRESULT GetClassInstance(  
    ID3DX11EffectClassInstanceVariable **ppEffectClassInstance  
>;
```

Parameters

ppEffectClassInstance

Type: [ID3DX11EffectClassInstanceVariable**](#)

Pointer to an [ID3DX11EffectClassInstanceVariable](#) pointer that will be set to the class instance.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectInterfaceVariable](#)

ID3DX11EffectInterfaceVariable::SetClassInstance method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Sets a class instance.

Syntax

```
HRESULT SetClassInstance(  
    ID3DX11EffectClassInstanceVariable *pEffectClassInstance  
>;
```

Parameters

pEffectClassInstance

Type: [ID3DX11EffectClassInstanceVariable*](#)

Pointer to an [ID3DX11EffectClassInstanceVariable](#) interface.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectInterfaceVariable](#)

ID3DX11EffectMatrixVariable interface

2/22/2020 • 2 minutes to read • [Edit Online](#)

A matrix-variable interface accesses a matrix.

Members

The **ID3DX11EffectMatrixVariable** interface inherits from [ID3DX11EffectVariable](#).

ID3DX11EffectMatrixVariable also has these types of members:

- [Methods](#)

Methods

The **ID3DX11EffectMatrixVariable** interface has these methods.

METHOD	DESCRIPTION
GetMatrix	Get a matrix.
GetMatrixArray	Get an array of matrices.
GetMatrixTranspose	Transpose and get a floating-point matrix.
GetMatrixTransposeArray	Transpose and get an array of floating-point matrices.
SetMatrix	Set a floating-point matrix.
SetMatrixArray	Set an array of floating-point matrices.
SetMatrixTranspose	Transpose and set a floating-point matrix.
SetMatrixTransposeArray	Transpose and set an array of floating-point matrices.

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
--------	----------------

Library	N/A (An Effects 11 library is available online as shared source.)
---------	---

See also

[ID3DX11EffectVariable](#)

[Effects 11 Interfaces](#)

[D3DX Interfaces](#)

ID3DX11EffectMatrixVariable::GetMatrix method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get a matrix.

Syntax

```
HRESULT GetMatrix(  
    float *pData  
) ;
```

Parameters

pData

Type: **float***

A pointer to the first element in a matrix.

Return value

Type: **HRESULT**

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectMatrixVariable](#)

ID3DX11EffectMatrixVariable::GetMatrixArray method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get an array of matrices.

Syntax

```
HRESULT GetMatrixArray(  
    float *pData,  
    UINT  Offset,  
    UINT  Count  
>;
```

Parameters

pData

Type: **float***

A pointer to the first element of the first matrix in an array of matrices.

Offset

Type: **UINT**

The offset (in number of matrices) between the start of the array and the first matrix returned.

Count

Type: **UINT**

The number of matrices in the returned array.

Return value

Type: **HRESULT**

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectMatrixVariable](#)

ID3DX11EffectMatrixVariable::GetMatrixTranspose method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Transpose and get a floating-point matrix.

Syntax

```
HRESULT GetMatrixTranspose(  
    float *pData  
>;
```

Parameters

pData

Type: **float***

A pointer to the first element of a transposed matrix.

Return value

Type: **HRESULT**

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

Transposing a matrix will rearrange the data order from row-column order to column-row order (or vice versa).

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

ID3DX11EffectMatrixVariable

ID3DX11EffectMatrixVariable::GetMatrixTransposeArray method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Transpose and get an array of floating-point matrices.

Syntax

```
HRESULT GetMatrixTransposeArray(  
    float *pData,  
    UINT  Offset,  
    UINT  Count  
>;
```

Parameters

pData

Type: **float***

A pointer to the first element of an array of tranposed matrices.

Offset

Type: **UINT**

The offset (in number of matrices) between the start of the array and the first matrix to get.

Count

Type: **UINT**

The number of matrices in the array to get.

Return value

Type: **HRESULT**

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

Transposing a matrix will rearrange the data order from row-column order to column-row order (or vice versa).

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectMatrixVariable](#)

ID3DX11EffectMatrixVariable::SetMatrix method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Set a floating-point matrix.

Syntax

```
HRESULT SetMatrix(  
    float *pData  
>;
```

Parameters

pData

Type: **float***

A pointer to the first element in the matrix.

Return value

Type: **HRESULT**

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectMatrixVariable](#)

ID3DX11EffectMatrixVariable::SetMatrixArray method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Set an array of floating-point matrices.

Syntax

```
HRESULT SetMatrixArray(  
    float *pData,  
    UINT  Offset,  
    UINT  Count  
>;
```

Parameters

pData

Type: **float***

A pointer to the first matrix.

Offset

Type: **UINT**

The number of matrix elements to skip from the start of the array.

Count

Type: **UINT**

The number of elements to set.

Return value

Type: **HRESULT**

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectMatrixVariable](#)

ID3DX11EffectMatrixVariable::SetMatrixTranspose method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Transpose and set a floating-point matrix.

Syntax

```
HRESULT SetMatrixTranspose(  
    float *pData  
>;
```

Parameters

pData

Type: **float***

A pointer to the first element of a matrix.

Return value

Type: **HRESULT**

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

Transposing a matrix will rearrange the data order from row-column order to column-row order (or vice versa).

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

ID3DX11EffectMatrixVariable

ID3DX11EffectMatrixVariable::SetMatrixTransposeArray method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Transpose and set an array of floating-point matrices.

Syntax

```
HRESULT SetMatrixTransposeArray(  
    float *pData,  
    UINT  Offset,  
    UINT  Count  
>;
```

Parameters

pData

Type: **float***

A pointer to an array of matrices.

Offset

Type: **UINT**

The offset (in number of matrices) between the start of the array and the first matrix to set.

Count

Type: **UINT**

The number of matrices in the array to set.

Return value

Type: **HRESULT**

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

Transposing a matrix will rearrange the data order from row-column order to column-row order (or vice versa).

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectMatrixVariable](#)

ID3DX11EffectPass interface

2/22/2020 • 2 minutes to read • [Edit Online](#)

An **ID3DX11EffectPass** interface encapsulates state assignments within a technique.

The lifetime of an **ID3DX11EffectPass** object is equal to the lifetime of its parent **ID3DX11Effect** object.

- [Methods](#)

Methods

The **ID3DX11EffectPass** interface has these methods.

METHOD	DESCRIPTION
Apply	Set the state contained in a pass to the device.
ComputeStateBlockMask	Generate a mask for allowing/preventing state changes.
GetAnnotationByIndex	Get an annotation by index.
GetAnnotationByName	Get an annotation by name.
GetComputeShaderDesc	Get a compute-shader description.
GetDesc	Get a pass description.
GetDomainShaderDesc	Get a domain-shader description.
GetGeometryShaderDesc	Get a geometry-shader description.
GetHullShaderDesc	Get hull-shader description.
GetPixelShaderDesc	Get a pixel-shader description.
GetVertexShaderDesc	Get a vertex-shader description.
IsValid	Test a pass to see if it contains valid syntax.

Remarks

A pass is a block of code that sets render-state objects and shaders. A pass is declared within a technique.

To get an effect-pass interface, call a method like [ID3DX11EffectTechnique::GetPassByName](#).

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[Effects 11 Interfaces](#)

[D3DX Interfaces](#)

ID3DX11EffectPass::Apply method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Set the state contained in a pass to the device.

Syntax

```
HRESULT Apply(  
    UINT           Flags,  
    ID3D11DeviceContext *pContext  
>;
```

Parameters

Flags

Type: [UINT](#)

Unused.

pContext

Type: [ID3D11DeviceContext*](#)

The [ID3D11DeviceContext](#) to apply the pass to.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectPass](#)

ID3DX11EffectPass::ComputeStateBlockMask method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Generate a mask for allowing/preventing state changes.

Syntax

```
HRESULT ComputeStateBlockMask(  
    D3DX11_STATE_BLOCK_MASK *pStateBlockMask  
>;
```

Parameters

pStateBlockMask

Type: [D3DX11_STATE_BLOCK_MASK*](#)

A pointer to a state-block mask (see [D3DX11_STATE_BLOCK_MASK](#)).

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectPass](#)

ID3DX11EffectPass::GetAnnotationByIndex method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get an annotation by index.

Syntax

```
ID3DX11EffectVariable* GetAnnotationByIndex(  
    UINT Index  
) ;
```

Parameters

Index

Type: [UINT](#)

A zero-based index.

Return value

Type: [ID3DX11EffectVariable*](#)

A pointer to an [ID3DX11EffectVariable](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectPass](#)

ID3DX11EffectPass::GetAnnotationByName method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get an annotation by name.

Syntax

```
ID3DX11EffectVariable* GetAnnotationByName(  
    LPCSTR Name  
) ;
```

Parameters

Name

Type: [LPCSTR](#)

The name of the annotation.

Return value

Type: [ID3DX11EffectVariable*](#)

A pointer to an [ID3DX11EffectVariable](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectPass](#)

ID3DX11EffectPass::GetComputeShaderDesc method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get a compute-shader description.

Syntax

```
HRESULT GetComputeShaderDesc(  
    D3DX11_PASS_SHADER_DESC *pDesc  
)
```

Parameters

pDesc

Type: [D3DX11_PASS_SHADER_DESC*](#)

A pointer to a compute-shader description (see [D3DX11_PASS_SHADER_DESC](#)).

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectPass](#)

ID3DX11EffectPass::GetDesc method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get a pass description.

Syntax

```
HRESULT GetDesc(  
    D3DX11_PASS_DESC *pDesc  
) ;
```

Parameters

pDesc

Type: [D3DX11_PASS_DESC*](#)

A pointer to a pass description (see [D3DX11_PASS_DESC](#)).

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

A pass is a block of code that sets render state and shaders (which in turn sets constant buffers, samplers and textures). An effect technique contains one or more passes.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectPass](#)

ID3DX11EffectPass::GetDomainShaderDesc method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get a domain-shader description.

Syntax

```
HRESULT GetDomainShaderDesc(  
    D3DX11_PASS_SHADER_DESC *pDesc  
)
```

Parameters

pDesc

Type: [D3DX11_PASS_SHADER_DESC*](#)

A pointer to a domain-shader description (see [D3DX11_PASS_SHADER_DESC](#)).

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectPass](#)

ID3DX11EffectPass::GetGeometryShaderDesc method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get a geometry-shader description.

Syntax

```
HRESULT GetGeometryShaderDesc(  
    D3DX11_PASS_SHADER_DESC *pDesc  
>;
```

Parameters

pDesc

Type: [D3DX11_PASS_SHADER_DESC*](#)

A pointer to a geometry-shader description (see [D3DX11_PASS_SHADER_DESC](#)).

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

An effect pass can contain render state assignments and shader object assignments.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

ID3DX11EffectPass

ID3DX11EffectPass::GetHullShaderDesc method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get hull-shader description.

Syntax

```
HRESULT GetHullShaderDesc(  
    D3DX11_PASS_SHADER_DESC *pDesc  
)
```

Parameters

pDesc

Type: [D3DX11_PASS_SHADER_DESC*](#)

A pointer to a hull-shader description (see [D3DX11_PASS_SHADER_DESC](#)).

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectPass](#)

ID3DX11EffectPass::GetPixelShaderDesc method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get a pixel-shader description.

Syntax

```
HRESULT GetPixelShaderDesc(  
    D3DX11_PASS_SHADER_DESC *pDesc  
)
```

Parameters

pDesc

Type: [D3DX11_PASS_SHADER_DESC*](#)

A pointer to a pixel-shader description (see [D3DX11_PASS_SHADER_DESC](#)).

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

An effect pass can contain render state assignments and shader object assignments.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectPass](#)

ID3DX11EffectPass::GetVertexShaderDesc method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get a vertex-shader description.

Syntax

```
HRESULT GetVertexShaderDesc(  
    D3DX11_PASS_SHADER_DESC *pDesc  
)
```

Parameters

pDesc

Type: [D3DX11_PASS_SHADER_DESC*](#)

A pointer to a vertex-shader description (see [D3DX11_PASS_SHADER_DESC](#)).

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

An effect pass can contain render state assignments and shader object assignments.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectPass](#)

ID3DX11EffectPass::IsValid method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Test a pass to see if it contains valid syntax.

Syntax

```
BOOL IsValid();
```

Parameters

This method has no parameters.

Return value

Type: **BOOL**

TRUE if the code syntax is valid; otherwise FALSE.

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectPass](#)

ID3DX11EffectRasterizerVariable interface

2/22/2020 • 2 minutes to read • [Edit Online](#)

A rasterizer-variable interface accesses rasterizer state.

Members

The **ID3DX11EffectRasterizerVariable** interface inherits from [ID3DX11EffectVariable](#).

ID3DX11EffectRasterizerVariable also has these types of members:

- [Methods](#)

Methods

The **ID3DX11EffectRasterizerVariable** interface has these methods.

METHOD	DESCRIPTION
GetBackingStore	Get a pointer to a variable that contains rasteriser state.
GetRasterizerState	Get a pointer to a rasterizer interface.
SetRasterizerState	Sets the rasterizer state.
UndoSetRasterizerState	Reverts a previously set rasterizer state.

Remarks

An [ID3DX11EffectVariable](#) interface is created when an effect is read into memory.

Effect variables are saved in memory in the backing store; when a technique is applied, the values in the backing store are copied to the device. You can use either of these methods to return state.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

ID3DX11EffectVariable

Effects 11 Interfaces

D3DX Interfaces

ID3DX11EffectRasterizerVariable::GetBackingStore method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get a pointer to a variable that contains rasteriser state.

Syntax

```
HRESULT GetBackingStore(
    UINT             Index,
    D3D11_RASTERIZER_DESC *pRasterizerDesc
);
```

Parameters

Index

Type: [UINT](#)

Index into an array of rasteriser-state descriptions. If there is only one rasteriser variable in the effect, use 0.

pRasterizerDesc

Type: [D3D11_RASTERIZER_DESC*](#)

A pointer to a rasteriser-state description (see [D3D11_RASTERIZER_DESC](#)).

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

Effect variables are saved in memory in the backing store; when a technique is applied, the values in the backing store are copied to the device. Backing store data can be used to recreate the variable when necessary.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
--------	----------------

Library	N/A (An Effects 11 library is available online as shared source.)
---------	---

See also

[ID3DX11EffectRasterizerVariable](#)

ID3DX11EffectRasterizerVariable::GetRasterizerState method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get a pointer to a rasterizer interface.

Syntax

```
HRESULT GetRasterizerState(  
    UINT           Index,  
    ID3D11RasterizerState **ppRasterizerState  
>;
```

Parameters

Index

Type: [UINT](#)

Index into an array of rasterizer interfaces. If there is only one rasterizer interface, use 0.

ppRasterizerState

Type: [ID3D11RasterizerState**](#)

The address of a pointer to a rasterizer interface (see [ID3D11RasterizerState](#)).

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h

Library	N/A (An Effects 11 library is available online as shared source.)
---------	---

See also

[ID3DX11EffectRasterizerVariable](#)

ID3DX11EffectRasterizerVariable::SetRasterizerState method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Sets the rasterizer state.

Syntax

```
HRESULT SetRasterizerState(  
    UINT             Index,  
    ID3D11RasterizerState *pRasterizerState  
>;
```

Parameters

Index

Type: [UINT](#)

Index into an array of rasterizer interfaces. If there is only one rasterizer interface, use 0.

pRasterizerState

Type: [ID3D11RasterizerState*](#)

Pointer to an [ID3D11RasterizerState](#) interface.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header

D3dx11effect.h

Library	N/A (An Effects 11 library is available online as shared source.)
---------	---

See also

[ID3DX11EffectRasterizerVariable](#)

ID3DX11EffectRasterizerVariable::UndoSetRasterizerState method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Reverts a previously set rasterizer state.

Syntax

```
HRESULT UndoSetRasterizerState(  
    UINT Index  
>;
```

Parameters

Index

Type: [UINT](#)

Index into an array of rasterizer interfaces. If there is only one rasterizer interface, use 0.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectRasterizerVariable](#)

ID3DX11EffectRenderTargetViewVariable interface

2/22/2020 • 2 minutes to read • [Edit Online](#)

A render-target-view interface accesses a render target.

Members

The ID3DX11EffectRenderTargetViewVariable interface inherits from [ID3DX11EffectRasterizerVariable](#). ID3DX11EffectRenderTargetViewVariable also has these types of members:

- [Methods](#)

Methods

The ID3DX11EffectRenderTargetViewVariable interface has these methods.

METHOD	DESCRIPTION
GetRenderTarget	Get a render-target.
GetRenderTargetArray	Get an array of render-targets.
SetRenderTarget	Set a render-target.
SetRenderTargetArray	Set an array of render-targets.

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectRasterizerVariable](#)

[Effects 11 Interfaces](#)

ID3DX11EffectRenderTargetViewVariable::GetRenderTarget method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get a render-target.

Syntax

```
HRESULT GetRenderTarget(  
    ID3D11RenderTargetView **ppResource  
) ;
```

Parameters

ppResource

Type: [ID3D11RenderTargetView**](#)

The address of a pointer to a render-target-view interface. See [ID3D11RenderTargetView](#).

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectRenderTargetViewVariable](#)

ID3DX11EffectRenderTargetViewVariable::GetRenderTargetArray method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get an array of render-targets.

Syntax

```
HRESULT GetRenderTargetArray(  
    ID3D11RenderTargetView **ppResources,  
    UINT             Offset,  
    UINT             Count  
)
```

Parameters

ppResources

Type: [ID3D11RenderTargetView**](#)

A pointer to an array of render-target-view interfaces. See [ID3D11RenderTargetView](#).

Offset

Type: [UINT](#)

The zero-based array index to get the first interface.

Count

Type: [UINT](#)

The number of elements in the array.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectRenderTargetViewVariable](#)

ID3DX11EffectRenderTargetViewVariable::SetRenderTarget method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Set a render-target.

Syntax

```
HRESULT SetRenderTarget(  
    ID3D11RenderTargetView *pResource  
>;
```

Parameters

pResource

Type: [ID3D11RenderTargetView*](#)

A pointer to a render-target-view interface. See [ID3D11RenderTargetView](#).

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectRenderTargetViewVariable](#)

ID3DX11EffectRenderTargetViewVariable::SetRenderTargetArray method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Set an array of render-targets.

Syntax

```
HRESULT SetRenderTargetArray(  
    ID3D11RenderTargetView **ppResources,  
    UINT             Offset,  
    UINT             Count  
>;
```

Parameters

ppResources

Type: [ID3D11RenderTargetView**](#)

Set an array of render-target-view interfaces. See [ID3D11RenderTargetView](#).

Offset

Type: [UINT](#)

The zero-based array index to store the first interface.

Count

Type: [UINT](#)

The number of elements in the array.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
--------	----------------

Library	N/A (An Effects 11 library is available online as shared source.)
---------	---

See also

[ID3DX11EffectRenderTargetViewVariable](#)

ID3DX11EffectSamplerVariable interface

2/22/2020 • 2 minutes to read • [Edit Online](#)

A sampler interface accesses sampler state.

Members

The ID3DX11EffectSamplerVariable interface inherits from [ID3DX11EffectVariable](#).

ID3DX11EffectSamplerVariable also has these types of members:

- [Methods](#)

Methods

The ID3DX11EffectSamplerVariable interface has these methods.

METHOD	DESCRIPTION
GetBackingStore	Get a pointer to a variable that contains sampler state.
GetSampler	Get a pointer to a sampler interface.
SetSampler	Set sampler state.
UndoSetSampler	Revert a previously set sampler state.

Remarks

An [ID3DX11EffectVariable](#) interface is created when an effect is read into memory.

Effect variables are saved in memory in the backing store; when a technique is applied, the values in the backing store are copied to the device. You can use either of these methods to return state.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

ID3DX11EffectVariable

Effects 11 Interfaces

D3DX Interfaces

ID3DX11EffectSamplerVariable::GetBackingStore method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get a pointer to a variable that contains sampler state.

Syntax

```
HRESULT GetBackingStore(  
    UINT             Index,  
    D3D11_SAMPLER_DESC *pSamplerDesc  
>;
```

Parameters

Index

Type: [UINT](#)

Index into an array of sampler descriptions. If there is only one sampler variable in the effect, use 0.

pSamplerDesc

Type: [D3D11_SAMPLER_DESC*](#)

A pointer to a sampler description (see [D3D11_SAMPLER_DESC](#)).

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h

Library	N/A (An Effects 11 library is available online as shared source.)
---------	---

See also

[ID3DX11EffectSamplerVariable](#)

ID3DX11EffectSamplerVariable::GetSampler method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get a pointer to a sampler interface.

Syntax

```
HRESULT GetSampler(  
    UINT             Index,  
    ID3D11SamplerState **ppSampler  
>;
```

Parameters

Index

Type: [UINT](#)

Index into an array of sampler interfaces. If there is only one sampler interface, use 0.

ppSampler

Type: [ID3D11SamplerState**](#)

The address of a pointer to a sampler interface (see [ID3D11SamplerState](#)).

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectSamplerVariable](#)

ID3DX11EffectSamplerVariable::SetSampler method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Set sampler state.

Syntax

```
HRESULT SetSampler(  
    UINT             Index,  
    ID3D11SamplerState *pSampler  
);
```

Parameters

Index

Type: **UINT**

Index into an array of sampler interfaces. If there is only one sampler interface, use 0.

pSampler

Type: **ID3D11SamplerState***

Pointer to an **ID3D11SamplerState** interface containing the sampler state.

Return value

Type: **HRESULT**

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectSamplerVariable](#)

ID3DX11EffectSamplerVariable::UndoSetSampler method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Revert a previously set sampler state.

Syntax

```
HRESULT UndoSetSampler(  
    UINT Index  
)
```

Parameters

Index

Type: **UINT**

Index into an array of sampler interfaces. If there is only one sampler interface, use 0.

Return value

Type: **HRESULT**

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectSamplerVariable](#)

ID3DX11EffectScalarVariable interface

2/22/2020 • 2 minutes to read • [Edit Online](#)

An effect-scalar-variable interface accesses scalar values.

Members

The **ID3DX11EffectScalarVariable** interface inherits from [ID3DX11EffectVariable](#).

ID3DX11EffectScalarVariable also has these types of members:

- [Methods](#)

Methods

The **ID3DX11EffectScalarVariable** interface has these methods.

METHOD	DESCRIPTION
GetBool	Get a boolean variable.
GetBoolArray	Get an array of boolean variables.
GetFloat	Get a floating-point variable.
GetFloatArray	Get an array of floating-point variables.
GetInt	Get an integer variable.
GetIntArray	Get an array of integer variables.
SetBool	Set a boolean variable.
SetBoolArray	Set an array of boolean variables.
SetFloat	Set a floating-point variable.
SetFloatArray	Set an array of floating-point variables.
SetInt	Set an integer variable.
SetIntArray	Set an array of integer variables.

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVariable](#)

[Effects 11 Interfaces](#)

[D3DX Interfaces](#)

ID3DX11EffectScalarVariable::GetBool method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get a boolean variable.

Syntax

```
HRESULT GetBool(  
    BOOL *pValue  
>;
```

Parameters

pValue

Type: **BOOL***

A pointer to the variable.

Return value

Type: **HRESULT**

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectScalarVariable](#)

ID3DX11EffectScalarVariable::GetBoolArray method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get an array of boolean variables.

Syntax

```
HRESULT GetBoolArray(  
    BOOL *pData,  
    UINT Offset,  
    UINT Count  
>;
```

Parameters

pData

Type: **BOOL***

A pointer to the start of the data to set.

Offset

Type: **UINT**

Must be set to 0; this is reserved for future use.

Count

Type: **UINT**

The number of array elements to set.

Return value

Type: **HRESULT**

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
--------	----------------

Library	N/A (An Effects 11 library is available online as shared source.)
---------	---

See also

[ID3DX11EffectScalarVariable](#)

ID3DX11EffectScalarVariable::GetFloat method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get a floating-point variable.

Syntax

```
HRESULT GetFloat(  
    float *pValue  
) ;
```

Parameters

pValue

Type: **float***

A pointer to the variable.

Return value

Type: **HRESULT**

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectScalarVariable](#)

ID3DX11EffectScalarVariable::GetFloatArray method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get an array of floating-point variables.

Syntax

```
HRESULT GetFloatArray(  
    float *pData,  
    UINT  Offset,  
    UINT  Count  
) ;
```

Parameters

pData

Type: **float***

A pointer to the start of the data to set.

Offset

Type: **UINT**

Must be set to 0; this is reserved for future use.

Count

Type: **UINT**

The number of array elements to set.

Return value

Type: **HRESULT**

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
--------	----------------

Library	N/A (An Effects 11 library is available online as shared source.)
---------	---

See also

[ID3DX11EffectScalarVariable](#)

ID3DX11EffectScalarVariable::GetInt method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get an integer variable.

Syntax

```
HRESULT GetInt(  
    int *pValue  
) ;
```

Parameters

pValue

Type: [int*](#)

A pointer to the variable.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectScalarVariable](#)

ID3DX11EffectScalarVariable::GetIntArray method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get an array of integer variables.

Syntax

```
HRESULT GetIntArray(  
    int *pData,  
    UINT Offset,  
    UINT Count  
>;
```

Parameters

pData

Type: [int*](#)

A pointer to the start of the data to set.

Offset

Type: [UINT](#)

Must be set to 0; this is reserved for future use.

Count

Type: [UINT](#)

The number of array elements to set.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header

D3dx11effect.h

Library	N/A (An Effects 11 library is available online as shared source.)
---------	---

See also

[ID3DX11EffectScalarVariable](#)

ID3DX11EffectScalarVariable::SetBool method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Set a boolean variable.

Syntax

```
HRESULT SetBool(  
    BOOL Value  
)
```

Parameters

Value

Type: [BOOL](#)

A pointer to the variable.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectScalarVariable](#)

ID3DX11EffectScalarVariable::SetBoolArray method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Set an array of boolean variables.

Syntax

```
HRESULT SetBoolArray(  
    BOOL *pData,  
    UINT Offset,  
    UINT Count  
>;
```

Parameters

pData

Type: **BOOL***

A pointer to the start of the data to set.

Offset

Type: **UINT**

Must be set to 0; this is reserved for future use.

Count

Type: **UINT**

The number of array elements to set.

Return value

Type: **HRESULT**

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
--------	----------------

Library	N/A (An Effects 11 library is available online as shared source.)
---------	---

See also

[ID3DX11EffectScalarVariable](#)

ID3DX11EffectScalarVariable::SetFloat method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Set a floating-point variable.

Syntax

```
HRESULT SetFloat(  
    float Value  
) ;
```

Parameters

Value

Type: **float**

A pointer to the variable.

Return value

Type: **HRESULT**

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectScalarVariable](#)

ID3DX11EffectScalarVariable::SetFloatArray method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Set an array of floating-point variables.

Syntax

```
HRESULT SetFloatArray(  
    float *pData,  
    UINT  Offset,  
    UINT  Count  
) ;
```

Parameters

pData

Type: **float***

A pointer to the start of the data to set.

Offset

Type: **UINT**

Must be set to 0; this is reserved for future use.

Count

Type: **UINT**

The number of array elements to set.

Return value

Type: **HRESULT**

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header

D3dx11effect.h

Library	N/A (An Effects 11 library is available online as shared source.)
---------	---

See also

[ID3DX11EffectScalarVariable](#)

ID3DX11EffectScalarVariable::SetInt method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Set an integer variable.

Syntax

```
HRESULT SetInt(  
    int Value  
)
```

Parameters

Value

Type: [int](#)

A pointer to the variable.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectScalarVariable](#)

ID3DX11EffectScalarVariable::SetIntArray method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Set an array of integer variables.

Syntax

```
HRESULT SetIntArray(  
    int *pData,  
    UINT Offset,  
    UINT Count  
) ;
```

Parameters

pData

Type: [int*](#)

A pointer to the start of the data to set.

Offset

Type: [UINT](#)

Must be set to 0; this is reserved for future use.

Count

Type: [UINT](#)

The number of array elements to set.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header

D3dx11effect.h

Library	N/A (An Effects 11 library is available online as shared source.)
---------	---

See also

[ID3DX11EffectScalarVariable](#)

ID3DX11EffectShaderResourceVariable interface

2/22/2020 • 2 minutes to read • [Edit Online](#)

A shader-resource interface accesses a shader resource.

Members

The **ID3DX11EffectShaderResourceVariable** interface inherits from [ID3DX11EffectVariable](#).

ID3DX11EffectShaderResourceVariable also has these types of members:

- [Methods](#)

Methods

The **ID3DX11EffectShaderResourceVariable** interface has these methods.

METHOD	DESCRIPTION
GetResource	Get a shader resource.
GetResourceArray	Get an array of shader resources.
SetResource	Set a shader resource.
SetResourceArray	Set an array of shader resources.

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVariable](#)

[Effects 11 Interfaces](#)

ID3DX11EffectShaderResourceVariable::GetResource method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get a shader resource.

Syntax

```
HRESULT GetResource(  
    ID3D11ShaderResourceView **ppResource  
>;
```

Parameters

ppResource

Type: [ID3D11ShaderResourceView**](#)

The address of a pointer to a shader-resource-view interface. See [ID3D11ShaderResourceView](#).

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectShaderResourceVariable](#)

ID3DX11EffectShaderResourceVariable::GetResourceArray method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get an array of shader resources.

Syntax

```
HRESULT GetResourceArray(  
    ID3D11ShaderResourceView **ppResources,  
    UINT             Offset,  
    UINT             Count  
) ;
```

Parameters

ppResources

Type: [ID3D11ShaderResourceView**](#)

The address of an array of shader-resource-view interfaces. See [ID3D11ShaderResourceView](#).

Offset

Type: [UINT](#)

The zero-based array index to get the first interface.

Count

Type: [UINT](#)

The number of elements in the array.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectShaderResourceVariable](#)

ID3DX11EffectShaderResourceVariable::SetResource method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Set a shader resource.

Syntax

```
HRESULT SetResource(  
    ID3D11ShaderResourceView *pResource  
>;
```

Parameters

pResource

Type: [ID3D11ShaderResourceView*](#)

The address of a pointer to a shader-resource-view interface. See [ID3D11ShaderResourceView](#).

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectShaderResourceVariable](#)

ID3DX11EffectShaderResourceVariable::SetResourceArray method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Set an array of shader resources.

Syntax

```
HRESULT SetResourceArray(  
    ID3D11ShaderResourceView **ppResources,  
    UINT             Offset,  
    UINT             Count  
) ;
```

Parameters

ppResources

Type: [ID3D11ShaderResourceView**](#)

The address of an array of shader-resource-view interfaces. See [ID3D11ShaderResourceView](#).

Offset

Type: [UINT](#)

The zero-based array index to get the first interface.

Count

Type: [UINT](#)

The number of elements in the array.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectShaderResourceVariable](#)

ID3DX11EffectShaderVariable interface

2/22/2020 • 2 minutes to read • [Edit Online](#)

A shader-variable interface accesses a shader variable.

Members

The **ID3DX11EffectShaderVariable** interface inherits from [ID3DX11EffectVariable](#).

ID3DX11EffectShaderVariable also has these types of members:

- [Methods](#)

Methods

The **ID3DX11EffectShaderVariable** interface has these methods.

METHOD	DESCRIPTION
GetComputeShader	Get a compute shader.
GetDomainShader	Get a domain shader.
GetGeometryShader	Get a geometry shader.
GetHullShader	Get a hull shader.
GetInputSignatureElementDesc	Get an input-signature description.
GetOutputSignatureElementDesc	Get an output-signature description.
GetPatchConstantSignatureElementDesc	Get a patch constant signature description.
GetPixelShader	Get a pixel shader.
GetShaderDesc	Get a shader description.
GetVertexShader	Get a vertex shader.

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVariable](#)

[Effects 11 Interfaces](#)

[D3DX Interfaces](#)

ID3DX11EffectShaderVariable::GetComputeShader method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get a compute shader.

Syntax

```
HRESULT GetComputeShader(  
    UINT             ShaderIndex,  
    ID3D11ComputeShader **ppPS  
>;
```

Parameters

ShaderIndex

Type: [UINT](#)

Index of the compute shader.

ppPS

Type: [ID3D11ComputeShader**](#)

Pointer to an [ID3D11ComputeShader](#) pointer that will be set to the compute shader on return.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
--------	----------------

Library	N/A (An Effects 11 library is available online as shared source.)
---------	---

See also

[ID3DX11EffectShaderVariable](#)

ID3DX11EffectShaderVariable::GetDomainShader method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get a domain shader.

Syntax

```
HRESULT GetDomainShader(  
    UINT             ShaderIndex,  
    ID3D11DomainShader **ppPS  
>;
```

Parameters

ShaderIndex

Type: [UINT](#)

Index of the domain shader.

ppPS

Type: [ID3D11DomainShader**](#)

Pointer to an [ID3D11DomainShader](#) pointer that will be set to the domain shader on return.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h

Library	N/A (An Effects 11 library is available online as shared source.)
---------	---

See also

[ID3DX11EffectShaderVariable](#)

ID3DX11EffectShaderVariable::GetGeometryShader method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get a geometry shader.

Syntax

```
HRESULT GetGeometryShader(  
    UINT             ShaderIndex,  
    ID3D11GeometryShader **ppGS  
>;
```

Parameters

ShaderIndex

Type: [UINT](#)

A zero-based index.

ppGS

Type: [ID3D11GeometryShader**](#)

A pointer to an [ID3D11GeometryShader](#) pointer that will be set to the geometry shader on return.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h

Library	N/A (An Effects 11 library is available online as shared source.)
---------	---

See also

[ID3DX11EffectShaderVariable](#)

ID3DX11EffectShaderVariable::GetHullShader method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get a hull shader.

Syntax

```
HRESULT GetHullShader(  
    UINT           ShaderIndex,  
    ID3D11HullShader **ppPS  
>;
```

Parameters

ShaderIndex

Type: [UINT](#)

Index of the shader.

ppPS

Type: [ID3D11HullShader**](#)

A pointer to an [ID3D11HullShader](#) pointer that will be set to the hull shader on return.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h

Library	N/A (An Effects 11 library is available online as shared source.)
---------	---

See also

[ID3DX11EffectShaderVariable](#)

ID3DX11EffectShaderVariable::GetInputSignatureElementDesc method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get an input-signature description.

Syntax

```
HRESULT GetInputSignatureElementDesc(
    UINT             ShaderIndex,
    UINT             Element,
    D3D11_SIGNATURE_PARAMETER_DESC *pDesc
);
```

Parameters

ShaderIndex

Type: **UINT**

A zero-based shader index.

Element

Type: **UINT**

A zero-based shader-element index.

pDesc

Type: **D3D11_SIGNATURE_PARAMETER_DESC***

A pointer to a parameter description (see [D3D11_SIGNATURE_PARAMETER_DESC](#)).

Return value

Type: **HRESULT**

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

An effect contains one or more shaders; each shader has an input and output signature; each signature contains one or more elements (or parameters). The shader index identifies the shader and the element index identifies the element (or parameter) in the shader signature.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectShaderVariable](#)

ID3DX11EffectShaderVariable::GetOutputSignatureElementDesc method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get an output-signature description.

Syntax

```
HRESULT GetOutputSignatureElementDesc(  
    UINT             ShaderIndex,  
    UINT             Element,  
    D3D11_SIGNATURE_PARAMETER_DESC *pDesc  
>;
```

Parameters

ShaderIndex

Type: [UINT](#)

A zero-based shader index.

Element

Type: [UINT](#)

A zero-based element index.

pDesc

Type: [D3D11_SIGNATURE_PARAMETER_DESC*](#)

A pointer to a parameter description (see [D3D11_SIGNATURE_PARAMETER_DESC](#)).

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

An effect contains one or more shaders; each shader has an input and output signature; each signature contains one or more elements (or parameters). The shader index identifies the shader and the element index identifies the element (or parameter) in the shader signature.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
--------	----------------

Library	N/A (An Effects 11 library is available online as shared source.)
---------	---

See also

[ID3DX11EffectShaderVariable](#)

ID3DX11EffectShaderVariable::GetPatchConstantSignatureElementDesc method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get a patch constant signature description.

Syntax

```
HRESULT GetPatchConstantSignatureElementDesc(
    UINT             ShaderIndex,
    UINT             Element,
    D3D11_SIGNATURE_PARAMETER_DESC *pDesc
);
```

Parameters

ShaderIndex

Type: [UINT](#)

A zero-based shader index.

Element

Type: [UINT](#)

A zero-based element index.

pDesc

Type: [D3D11_SIGNATURE_PARAMETER_DESC*](#)

A pointer to a parameter description (see [D3D11_SIGNATURE_PARAMETER_DESC](#)).

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectShaderVariable](#)

ID3DX11EffectShaderVariable::GetPixelShader method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get a pixel shader.

Syntax

```
HRESULT GetPixelShader(  
    UINT           ShaderIndex,  
    ID3D11PixelShader **ppPS  
>;
```

Parameters

ShaderIndex

Type: [UINT](#)

A zero-based index.

ppPS

Type: [ID3D11PixelShader**](#)

A pointer to an [ID3D11PixelShader](#) pointer that will be set to the pixel shader on return.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
--------	----------------

Library	N/A (An Effects 11 library is available online as shared source.)
---------	---

See also

[ID3DX11EffectShaderVariable](#)

ID3DX11EffectShaderVariable::GetShaderDesc method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get a shader description.

Syntax

```
HRESULT GetShaderDesc(  
    UINT           ShaderIndex,  
    D3DX11_EFFECT_SHADER_DESC *pDesc  
>;
```

Parameters

ShaderIndex

Type: [UINT](#)

A zero-based index.

pDesc

Type: [D3DX11_EFFECT_SHADER_DESC*](#)

A pointer to a shader description (see [D3DX11_EFFECT_SHADER_DESC](#)).

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h

Library	N/A (An Effects 11 library is available online as shared source.)
---------	---

See also

[ID3DX11EffectShaderVariable](#)

ID3DX11EffectShaderVariable::GetVertexShader method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get a vertex shader.

Syntax

```
HRESULT GetVertexShader(  
    UINT             ShaderIndex,  
    ID3D11VertexShader **ppVS  
>;
```

Parameters

ShaderIndex

Type: [UINT](#)

A zero-based index.

ppVS

Type: [ID3D11VertexShader**](#)

A pointer to an [ID3D11VertexShader](#) pointer that will be set to the vertex shader on return.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h

Library	N/A (An Effects 11 library is available online as shared source.)
---------	---

See also

[ID3DX11EffectShaderVariable](#)

ID3DX11EffectStringVariable interface

2/22/2020 • 2 minutes to read • [Edit Online](#)

A string-variable interface accesses a string variable.

Members

The ID3DX11EffectStringVariable interface inherits from [ID3DX11EffectVariable](#).

ID3DX11EffectStringVariable also has these types of members:

- [Methods](#)

Methods

The ID3DX11EffectStringVariable interface has these methods.

METHOD	DESCRIPTION
GetString	Get the string.
GetStringArray	Get an array of strings.

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVariable](#)

[Effects 11 Interfaces](#)

[D3DX Interfaces](#)

ID3DX11EffectStringVariable::GetString method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get the string.

Syntax

```
HRESULT GetString(  
    LPCSTR *ppString  
) ;
```

Parameters

ppString

Type: [LPCSTR*](#)

A pointer to the string.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectStringVariable](#)

ID3DX11EffectStringVariable::GetStringArray method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get an array of strings.

Syntax

```
HRESULT GetStringArray(  
    LPCSTR *ppStrings,  
    UINT    Offset,  
    UINT    Count  
>;
```

Parameters

ppStrings

Type: [LPCSTR*](#)

A pointer to the first string in the array.

Offset

Type: [UINT](#)

The offset (in number of strings) between the start of the array and the first string to get.

Count

Type: [UINT](#)

The number of strings in the returned array.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
--------	----------------

Library	N/A (An Effects 11 library is available online as shared source.)
---------	---

See also

[ID3DX11EffectStringVariable](#)

ID3DX11EffectTechnique interface

2/22/2020 • 2 minutes to read • [Edit Online](#)

An **ID3DX11EffectTechnique** interface is a collection of passes.

The lifetime of an **ID3DX11EffectTechnique** object is equal to the lifetime of its parent **ID3DX11Effect** object.

- [Methods](#)

Methods

The **ID3DX11EffectTechnique** interface has these methods.

METHOD	DESCRIPTION
ComputeStateBlockMask	Compute a state-block mask to allow/prevent state changes.
GetAnnotationByIndex	Get an annotation by index.
GetAnnotationByName	Get an annotation by name.
GetDesc	Get a technique description.
GetPassByIndex	Get a pass by index.
GetPassByName	Get a pass by name.
IsValid	Test a technique to see if it contains valid syntax.

Remarks

An effect contains one or more techniques; each technique contains one or more passes; each pass contains state assignments.

To get an effect-technique interface, call a method such as [ID3DX11Effect::GetTechniqueByName](#).

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[Effects 11 Interfaces](#)

[D3DX Interfaces](#)

ID3DX11EffectTechnique::ComputeStateBlockMask method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Compute a state-block mask to allow/prevent state changes.

Syntax

```
HRESULT ComputeStateBlockMask(  
    D3DX11_STATE_BLOCK_MASK *pStateBlockMask  
>;
```

Parameters

pStateBlockMask

Type: [D3DX11_STATE_BLOCK_MASK*](#)

A pointer to a state-block mask (see [D3DX11_STATE_BLOCK_MASK](#)).

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectTechnique](#)

ID3DX11EffectTechnique::GetAnnotationByIndex method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get an annotation by index.

Syntax

```
ID3DX11EffectVariable* GetAnnotationByIndex(  
    UINT Index  
)
```

Parameters

Index

Type: **UINT**

The zero-based index of the interface pointer.

Return value

Type: **ID3DX11EffectVariable***

A pointer to an **ID3DX11EffectVariable**.

Remarks

Use an annotation to attach a piece of metadata to a technique.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

ID3DX11EffectTechnique

ID3DX11EffectTechnique::GetAnnotationByName method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get an annotation by name.

Syntax

```
ID3DX11EffectVariable* GetAnnotationByName(  
    LPCSTR Name  
)
```

Parameters

Name

Type: [LPCSTR](#)

Name of the annotation.

Return value

Type: [ID3DX11EffectVariable*](#)

A pointer to an [ID3DX11EffectVariable](#).

Remarks

Use an annotation to attach a piece of metadata to a technique.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

ID3DX11EffectTechnique

ID3DX11EffectTechnique::GetDesc method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get a technique description.

Syntax

```
HRESULT GetDesc(  
    D3DX11_TECHNIQUE_DESC *pDesc  
>;
```

Parameters

pDesc

Type: [D3DX11_TECHNIQUE_DESC*](#)

A pointer to a technique description (see [D3DX11_TECHNIQUE_DESC](#)).

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectTechnique](#)

ID3DX11EffectTechnique::GetPassByIndex method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get a pass by index.

Syntax

```
ID3DX11EffectPass* GetPassByIndex(  
    UINT Index  
) ;
```

Parameters

Index

Type: **UINT**

A zero-based index.

Return value

Type: **ID3DX11EffectPass***

A pointer to a **ID3DX11EffectPass**.

Remarks

A technique contains one or more passes; get a pass using a name or an index.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectTechnique](#)

ID3DX11EffectTechnique::GetPassByName method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get a pass by name.

Syntax

```
ID3DX11EffectPass* GetPassByName(  
    LPCSTR Name  
) ;
```

Parameters

Name

Type: [LPCSTR](#)

The name of the pass.

Return value

Type: [ID3DX11EffectPass*](#)

A pointer to an [ID3DX11EffectPass](#).

Remarks

A technique contains one or more passes; get a pass using a name or an index.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectTechnique](#)

ID3DX11EffectTechnique::IsValid method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Test a technique to see if it contains valid syntax.

Syntax

```
BOOL IsValid();
```

Parameters

This method has no parameters.

Return value

Type: **BOOL**

TRUE if the code syntax is valid; otherwise FALSE.

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectTechnique](#)

ID3DX11EffectType interface

2/22/2020 • 2 minutes to read • [Edit Online](#)

The **ID3DX11EffectType** interface accesses effect variables by type.

The lifetime of an **ID3DX11EffectType** object is equal to the lifetime of its parent **ID3DX11Effect** object.

- [Methods](#)

Methods

The **ID3DX11EffectType** interface has these methods.

METHOD	DESCRIPTION
GetDesc	Get an effect-type description.
GetMemberName	Get the name of a member.
GetMemberSemantic	Get the semantic attached to a member.
GetMemberTypeByIndex	Get a member type by index.
GetMemberTypeByName	Get an member type by name.
GetMemberTypeBySemantic	Get a member type by semantic.
IsValid	Tests that the effect type is valid.

Remarks

To get information about an effect type from an effect variable, call [ID3DX11EffectVariable::GetType](#).

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[Effects 11 Interfaces](#)

[D3DX Interfaces](#)

ID3DX11EffectType::GetDesc method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get an effect-type description.

Syntax

```
HRESULT GetDesc(  
    D3DX11_EFFECT_TYPE_DESC *pDesc  
)
```

Parameters

pDesc

Type: [D3DX11_EFFECT_TYPE_DESC*](#)

A pointer to an effect-type description. See [D3DX11_EFFECT_TYPE_DESC](#).

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

The effect-variable description contains data about the name, annotations, semantic, flags and buffer offset of the effect type.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectType](#)

ID3DX11EffectType::GetMemberName method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get the name of a member.

Syntax

```
LPCSTR GetMemberName(  
    UINT Index  
) ;
```

Parameters

Index

Type: [UINT](#)

A zero-based index.

Return value

Type: [LPCSTR](#)

The name of the member.

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectType](#)

ID3DX11EffectType::GetMemberSemantic method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get the semantic attached to a member.

Syntax

```
LPCSTR GetMemberSemantic(  
    UINT Index  
) ;
```

Parameters

Index

Type: [UINT](#)

A zero-based index.

Return value

Type: [LPCSTR](#)

A string that contains the semantic.

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectType](#)

ID3DX11EffectType::GetMemberTypeByIndex method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get a member type by index.

Syntax

```
ID3DX11EffectType* GetMemberTypeByIndex(  
    UINT Index  
) ;
```

Parameters

Index

Type: [UINT](#)

A zero-based index.

Return value

Type: [ID3DX11EffectType*](#)

A pointer to an [ID3DX11EffectType](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectType](#)

ID3DX11EffectType::GetMemberTypeByName method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get an member type by name.

Syntax

```
ID3DX11EffectType* GetMemberTypeByName(  
    LPCSTR Name  
)
```

Parameters

Name

Type: [LPCSTR](#)

A member's name.

Return value

Type: [ID3DX11EffectType*](#)

A pointer to an [ID3DX11EffectType](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectType](#)

ID3DX11EffectType::GetMemberTypeBySemantic method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get a member type by semantic.

Syntax

```
ID3DX11EffectType* GetMemberTypeBySemantic(  
    LPCSTR Semantic  
);
```

Parameters

Semantic

Type: [LPCSTR](#)

A semantic.

Return value

Type: [ID3DX11EffectType*](#)

A pointer to an [ID3DX11EffectType](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectType](#)

ID3DX11EffectType::IsValid method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Tests that the effect type is valid.

Syntax

```
BOOL IsValid();
```

Parameters

This method has no parameters.

Return value

Type: **BOOL**

TRUE if it is valid; otherwise FALSE.

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectType](#)

ID3DX11EffectUnorderedAccessViewVariable interface

2/22/2020 • 2 minutes to read • [Edit Online](#)

Accesses an unordered access view.

Members

The ID3DX11EffectUnorderedAccessViewVariable interface inherits from [ID3DX11EffectVariable](#).
ID3DX11EffectUnorderedAccessViewVariable also has these types of members:

- [Methods](#)

Methods

The ID3DX11EffectUnorderedAccessViewVariable interface has these methods.

METHOD	DESCRIPTION
GetUnorderedAccessView	Get an unordered-access-view.
GetUnorderedAccessViewArray	Get an array of unordered-access-views.
SetUnorderedAccessView	Set an unordered-access-view.
SetUnorderedAccessViewArray	Set an array of unordered-access-views.

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVariable](#)

[Effects 11 Interfaces](#)

[D3DX Interfaces](#)

ID3DX11EffectUnorderedAccessViewVariable::GetUnorderedAccessView method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get an unordered-access-view.

Syntax

```
HRESULT GetUnorderedAccessView(  
    ID3D11UnorderedAccessView **ppResource  
) ;
```

Parameters

ppResource

Type: [ID3D11UnorderedAccessView**](#)

Pointer to an [ID3D11UnorderedAccessView](#) pointer that will be set on return.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectUnorderedAccessViewVariable](#)

ID3DX11EffectUnorderedAccessViewVariable::GetUnorderedAccessView method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get an array of unordered-access-views.

Syntax

```
HRESULT GetUnorderedAccessViewArray(
    ID3D11UnorderedAccessView **ppResources,
    UINT                 Offset,
    UINT                 Count
);
```

Parameters

ppResources

Type: [ID3D11UnorderedAccessView**](#)

Pointer to an [ID3D11UnorderedAccessView](#) pointer that will be set to the UAV array on return.

Offset

Type: [UINT](#)

Index of the first interface.

Count

Type: [UINT](#)

Number of elements in the array.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectUnorderedAccessViewVariable](#)

ID3DX11EffectUnorderedAccessViewVariable::SetUnorderedAccessView method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Set an unordered-access-view.

Syntax

```
HRESULT SetUnorderedAccessView(  
    ID3D11UnorderedAccessView *pResource  
) ;
```

Parameters

pResource

Type: [ID3D11UnorderedAccessView*](#)

Pointer to an [ID3D11UnorderedAccessView](#).

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectUnorderedAccessViewVariable](#)

ID3DX11EffectUnorderedAccessViewVariable::SetUnorderedAccessView method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Set an array of unordered-access-views.

Syntax

```
HRESULT SetUnorderedAccessViewArray(
    ID3D11UnorderedAccessView **ppResources,
    UINT                 Offset,
    UINT                 Count
);
```

Parameters

ppResources

Type: [ID3D11UnorderedAccessView**](#)

An array of [ID3D11UnorderedAccessView](#) pointers.

Offset

Type: [UINT](#)

Index of the first unordered-access-view.

Count

Type: [UINT](#)

Number of elements in the array.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectUnorderedAccessViewVariable](#)

ID3DX11EffectVariable interface

2/22/2020 • 2 minutes to read • [Edit Online](#)

The **ID3DX11EffectVariable** interface is the base class for all effect variables.

The lifetime of an **ID3DX11EffectVariable** object is equal to the lifetime of its parent **ID3DX11Effect** object.

- [Methods](#)

Methods

The **ID3DX11EffectVariable** interface has these methods.

METHOD	DESCRIPTION
AsBlend	Get a effect-blend variable.
AsClassInstance	Get a class-instance variable.
AsConstantBuffer	Get a constant buffer.
AsDepthStencil	Get a depth-stencil variable.
AsDepthStencilView	Get a depth-stencil-view variable.
AsInterface	Get an interface variable.
AsMatrix	Get a matrix variable.
AsRasterizer	Get a rasterizer variable.
AsRenderTargetView	Get a render-target-view variable.
AsSampler	Get a sampler variable.
AsScalar	Get a scalar variable.
AsShader	Get a shader variable.
AsShaderResource	Get a shader-resource variable.
AsString	Get a string variable.
AsUnorderedAccessView	Get an unordered-access-view variable.
AsVector	Get a vector variable.
GetAnnotationByIndex	Get an annotation by index.
GetAnnotationByName	Get an annotation by name.
GetDesc	Get a description.
GetElement	Get an array element.

METHOD	DESCRIPTION
GetMemberByIndex	Get a structure member by index.
GetMemberByName	Get a structure member by name.
GetMemberBySemantic	Get a structure member by semantic.
GetParentConstantBuffer	Get a constant buffer.
GetRawValue	Get data.
GetType	Get type information.
IsValid	Compare the data type with the data stored.
SetRawValue	Set data.

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[Effects 11 Interfaces](#)

[D3DX Interfaces](#)

ID3DX11EffectVariable::AsBlend method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get a effect-blend variable.

Syntax

```
ID3DX11EffectBlendVariable* AsBlend();
```

Parameters

This method has no parameters.

Return value

Type: [ID3DX11EffectBlendVariable*](#)

A pointer to an effect blend variable. See [ID3DX11EffectBlendVariable](#).

Remarks

AsBlend returns a version of the effect variable that has been specialized to an effect-blend variable. Similar to a cast, this specialization will return an invalid object if the effect variable does not contain effect-blend data.

Applications can test the returned object for validity by calling [IsValid](#).

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVariable](#)

ID3DX11EffectVariable::AsClassInstance method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get a class-instance variable.

Syntax

```
ID3DX11EffectClassInstanceVariable* AsClassInstance();
```

Parameters

This method has no parameters.

Return value

Type: [ID3DX11EffectClassInstanceVariable*](#)

A pointer to class-instance variable. See [ID3DX11EffectClassInstanceVariable](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVariable](#)

ID3DX11EffectVariable::AsConstantBuffer method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get a constant buffer.

Syntax

```
ID3DX11EffectConstantBuffer* AsConstantBuffer();
```

Parameters

This method has no parameters.

Return value

Type: [ID3DX11EffectConstantBuffer*](#)

A pointer to a constant buffer. See [ID3DX11EffectConstantBuffer](#).

Remarks

AsConstantBuffer returns a version of the effect variable that has been specialized to a constant buffer. Similar to a cast, this specialization will return an invalid object if the effect variable does not contain constant buffer data.

Applications can test the returned object for validity by calling [IsValid](#).

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVariable](#)

ID3DX11EffectVariable::AsDepthStencil method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get a depth-stencil variable.

Syntax

```
ID3DX11EffectDepthStencilVariable* AsDepthStencil();
```

Parameters

This method has no parameters.

Return value

Type: [ID3DX11EffectDepthStencilVariable*](#)

A pointer to a depth-stencil variable. See [ID3DX11EffectDepthStencilVariable](#).

Remarks

AsDepthStencil returns a version of the effect variable that has been specialized to a depth-stencil variable. Similar to a cast, this specialization will return an invalid object if the effect variable does not contain depth-stencil data.

Applications can test the returned object for validity by calling [IsValid](#).

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVariable](#)

ID3DX11EffectVariable::AsDepthStencilView method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get a depth-stencil-view variable.

Syntax

```
ID3DX11EffectDepthStencilViewVariable* AsDepthStencilView();
```

Parameters

This method has no parameters.

Return value

Type: [ID3DX11EffectDepthStencilViewVariable*](#)

A pointer to a depth-stencil-view variable. See [ID3DX11EffectDepthStencilViewVariable](#).

Remarks

This method returns a version of the effect variable that has been specialized to a depth-stencil-view variable. Similar to a cast, this specialization will return an invalid object if the effect variable does not contain depth-stencil-view data.

Applications can test the returned object for validity by calling [IsValid](#).

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVariable](#)

ID3DX11EffectVariable::AsInterface method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get an interface variable.

Syntax

```
ID3DX11EffectInterfaceVariable* AsInterface();
```

Parameters

This method has no parameters.

Return value

Type: [ID3DX11EffectInterfaceVariable*](#)

A pointer to an interface variable. See [ID3DX11EffectInterfaceVariable](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVariable](#)

ID3DX11EffectVariable::AsMatrix method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get a matrix variable.

Syntax

```
ID3DX11EffectMatrixVariable* AsMatrix();
```

Parameters

This method has no parameters.

Return value

Type: [ID3DX11EffectMatrixVariable*](#)

A pointer to a matrix variable. See [ID3DX11EffectMatrixVariable](#).

Remarks

AsMatrix returns a version of the effect variable that has been specialized to a matrix variable. Similar to a cast, this specialization will return an invalid object if the effect variable does not contain matrix data.

Applications can test the returned object for validity by calling [IsValid](#).

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVariable](#)

ID3DX11EffectVariable::AsRasterizer method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get a rasterizer variable.

Syntax

```
ID3DX11EffectRasterizerVariable* AsRasterizer();
```

Parameters

This method has no parameters.

Return value

Type: [ID3DX11EffectRasterizerVariable*](#)

A pointer to a rasterizer variable. See [ID3DX11EffectRasterizerVariable](#).

Remarks

AsRasterizer returns a version of the effect variable that has been specialized to a rasterizer variable. Similar to a cast, this specialization will return an invalid object if the effect variable does not contain rasterizer data.

Applications can test the returned object for validity by calling [IsValid](#).

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVariable](#)

ID3DX11EffectVariable::AsRenderTargetView method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get a render-target-view variable.

Syntax

```
ID3DX11EffectRenderTargetViewVariable* AsRenderTargetView();
```

Parameters

This method has no parameters.

Return value

Type: [ID3DX11EffectRenderTargetViewVariable*](#)

A pointer to a render-target-view variable. See [ID3DX11EffectRenderTargetViewVariable](#).

Remarks

This method returns a version of the effect variable that has been specialized to a render-target-view variable. Similar to a cast, this specialization will return an invalid object if the effect variable does not contain render-target-view data.

Applications can test the returned object for validity by calling [IsValid](#).

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVariable](#)

ID3DX11EffectVariable::AsSampler method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get a sampler variable.

Syntax

```
ID3DX11EffectSamplerVariable* AsSampler();
```

Parameters

This method has no parameters.

Return value

Type: [ID3DX11EffectSamplerVariable*](#)

A pointer to a sampler variable. See [ID3DX11EffectSamplerVariable](#).

Remarks

AsSampler returns a version of the effect variable that has been specialized to a sampler variable. Similar to a cast, this specialization will return an invalid object if the effect variable does not contain sampler data.

Applications can test the returned object for validity by calling [IsValid](#).

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVariable](#)

ID3DX11EffectVariable::AsScalar method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get a scalar variable.

Syntax

```
ID3DX11EffectScalarVariable* AsScalar();
```

Parameters

This method has no parameters.

Return value

Type: [ID3DX11EffectScalarVariable*](#)

A pointer to a scalar variable. See [ID3DX11EffectScalarVariable](#).

Remarks

AsScalar returns a version of the effect variable that has been specialized to a scalar variable. Similar to a cast, this specialization will return an invalid object if the effect variable does not contain scalar data.

Applications can test the returned object for validity by calling [IsValid](#).

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVariable](#)

ID3DX11EffectVariable::AsShader method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get a shader variable.

Syntax

```
ID3DX11EffectShaderVariable* AsShader();
```

Parameters

This method has no parameters.

Return value

Type: [ID3DX11EffectShaderVariable*](#)

A pointer to a shader variable. See [ID3DX11EffectShaderVariable](#).

Remarks

AsShader returns a version of the effect variable that has been specialized to a shader variable. Similar to a cast, this specialization will return an invalid object if the effect variable does not contain shader data.

Applications can test the returned object for validity by calling [IsValid](#).

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVariable](#)

ID3DX11EffectVariable::AsShaderResource method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get a shader-resource variable.

Syntax

```
ID3DX11EffectShaderResourceVariable* AsShaderResource();
```

Parameters

This method has no parameters.

Return value

Type: [ID3DX11EffectShaderResourceVariable*](#)

A pointer to a shader-resource variable. See [ID3DX11EffectShaderResourceVariable](#).

Remarks

AsShaderResource returns a version of the effect variable that has been specialized to a shader-resource variable. Similar to a cast, this specialization will return an invalid object if the effect variable does not contain shader-resource data.

Applications can test the returned object for validity by calling [IsValid](#).

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVariable](#)

ID3DX11EffectVariable::AsString method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get a string variable.

Syntax

```
ID3DX11EffectStringVariable* AsString();
```

Parameters

This method has no parameters.

Return value

Type: [ID3DX11EffectStringVariable*](#)

A pointer to a string variable. See [ID3DX11EffectStringVariable](#).

Remarks

AsString returns a version of the effect variable that has been specialized to a string variable. Similar to a cast, this specialization will return an invalid object if the effect variable does not contain string data.

Applications can test the returned object for validity by calling [IsValid](#).

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVariable](#)

ID3DX11EffectVariable::AsUnorderedAccessView method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get an unordered-access-view variable.

Syntax

```
ID3DX11EffectUnorderedAccessViewVariable* AsUnorderedAccessView();
```

Parameters

This method has no parameters.

Return value

Type: [ID3DX11EffectUnorderedAccessViewVariable*](#)

A pointer to an unordered-access-view variable. See [ID3DX11EffectUnorderedAccessViewVariable](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVariable](#)

ID3DX11EffectVariable::AsVector method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get a vector variable.

Syntax

```
ID3DX11EffectVectorVariable* AsVector();
```

Parameters

This method has no parameters.

Return value

Type: [ID3DX11EffectVectorVariable*](#)

A pointer to a vector variable. See [ID3DX11EffectVectorVariable](#).

Remarks

AsVector returns a version of the effect variable that has been specialized to a vector variable. Similar to a cast, this specialization will return an invalid object if the effect variable does not contain vector data.

Applications can test the returned object for validity by calling [IsValid](#).

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVariable](#)

ID3DX11EffectVariable::GetAnnotationByIndex method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get an annotation by index.

Syntax

```
ID3DX11EffectVariable* GetAnnotationByIndex(  
    UINT Index  
)
```

Parameters

Index

Type: [UINT](#)

A zero-based index.

Return value

Type: [ID3DX11EffectVariable*](#)

A pointer to an [ID3DX11EffectVariable](#).

Remarks

Annoations can be attached to a technique, a pass, or a global variable.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

ID3DX11EffectVariable

ID3DX11EffectVariable::GetAnnotationByName method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get an annotation by name.

Syntax

```
ID3DX11EffectVariable* GetAnnotationByName(  
    LPCSTR Name  
)
```

Parameters

Name

Type: [LPCSTR](#)

The annotation name.

Return value

Type: [ID3DX11EffectVariable*](#)

A pointer to an [ID3DX11EffectVariable](#). Note that if the annotation is not found the [ID3DX11EffectVariable](#) returned will be empty. The [ID3DX11EffectVariable::IsValid](#) method should be called to determine whether the annotation was found.

Remarks

Annotations can be attached to a technique, a pass, or a global variable.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVariable](#)

ID3DX11EffectVariable::GetDesc method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get a description.

Syntax

```
HRESULT GetDesc(  
    D3DX11_EFFECT_VARIABLE_DESC *pDesc  
) ;
```

Parameters

pDesc

Type: [D3DX11_EFFECT_VARIABLE_DESC*](#)

A pointer to an effect-variable description (see [D3DX11_EFFECT_VARIABLE_DESC](#)).

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVariable](#)

ID3DX11EffectVariable::GetElement method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get an array element.

Syntax

```
ID3DX11EffectVariable* GetElement(  
    UINT Index  
) ;
```

Parameters

Index

Type: **UINT**

A zero-based index; otherwise 0.

Return value

Type: **ID3DX11EffectVariable***

A pointer to an **ID3DX11EffectVariable**.

Remarks

If the effect variable is an array, use this method to return one of the elements.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVariable](#)

ID3DX11EffectVariable::GetMemberByIndex method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get a structure member by index.

Syntax

```
ID3DX11EffectVariable* GetMemberByIndex(  
    UINT Index  
) ;
```

Parameters

Index

Type: [UINT](#)

A zero-based index.

Return value

Type: [ID3DX11EffectVariable*](#)

A pointer to an [ID3DX11EffectVariable](#).

Remarks

If the effect variable is an structure, use this method to look up a member by index.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVariable](#)

ID3DX11EffectVariable::GetMemberByName method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get a structure member by name.

Syntax

```
ID3DX11EffectVariable* GetMemberByName(  
    LPCSTR Name  
) ;
```

Parameters

Name

Type: [LPCSTR](#)

Member name.

Return value

Type: [ID3DX11EffectVariable*](#)

A pointer to an [ID3DX11EffectVariable](#).

Remarks

If the effect variable is an structure, use this method to look up a member by name.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVariable](#)

ID3DX11EffectVariable::GetMemberBySemantic method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get a structure member by semantic.

Syntax

```
ID3DX11EffectVariable* GetMemberBySemantic(  
    LPCSTR Semantic  
);
```

Parameters

Semantic

Type: [LPCSTR](#)

The semantic.

Return value

Type: [ID3DX11EffectVariable*](#)

A pointer to an [ID3DX11EffectVariable](#).

Remarks

If the effect variable is an structure, use this method to look up a member by attached semantic.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

ID3DX11EffectVariable

ID3DX11EffectVariable::GetParentConstantBuffer method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get a constant buffer.

Syntax

```
ID3DX11EffectConstantBuffer* GetParentConstantBuffer();
```

Parameters

This method has no parameters.

Return value

Type: [ID3DX11EffectConstantBuffer*](#)

A pointer to a [ID3DX11EffectConstantBuffer](#).

Remarks

Effect variables are read-from or written-to a constant buffer.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVariable](#)

ID3DX11EffectVariable::GetRawValue method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get data.

Syntax

```
HRESULT GetRawValue(  
    void *pData,  
    UINT Offset,  
    UINT Count  
) ;
```

Parameters

pData

Type: **void***

A pointer to the variable.

Offset

Type: **UINT**

The offset (in bytes) from the beginning of the pointer to the data.

Count

Type: **UINT**

The number of bytes to get.

Return value

Type: **HRESULT**

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

This method does no conversion or type checking; it is therefore a very quick way to access array items.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVariable](#)

ID3DX11EffectVariable::GetType method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get type information.

Syntax

```
ID3DX11EffectType* GetType();
```

Parameters

This method has no parameters.

Return value

Type: [ID3DX11EffectType*](#)

A pointer to an [ID3DX11EffectType](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVariable](#)

ID3DX11EffectVariable::IsValid method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Compare the data type with the data stored.

Syntax

```
BOOL IsValid();
```

Parameters

This method has no parameters.

Return value

Type: **BOOL**

TRUE if the syntax is valid; otherwise **FALSE**.

Remarks

This method checks that the data type matches the data stored after casting one interface to another (using any of the As methods).

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVariable](#)

ID3DX11EffectVariable::SetRawValue method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Set data.

Syntax

```
HRESULT SetRawValue(  
    void *pData,  
    UINT Offset,  
    UINT Count  
) ;
```

Parameters

pData

Type: **void***

A pointer to the variable.

Offset

Type: **UINT**

The offset (in bytes) from the beginning of the pointer to the data.

Count

Type: **UINT**

The number of bytes to set.

Return value

Type: **HRESULT**

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

This method does no conversion or type checking; it is therefore a very quick way to access array items.

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVariable](#)

ID3DX11EffectVectorVariable interface

2/22/2020 • 2 minutes to read • [Edit Online](#)

A vector-variable interface accesses a four-component vector.

Members

The ID3DX11EffectVectorVariable interface inherits from [ID3DX11EffectVariable](#).

ID3DX11EffectVectorVariable also has these types of members:

- [Methods](#)

Methods

The ID3DX11EffectVectorVariable interface has these methods.

METHOD	DESCRIPTION
GetBoolVector	Get a four-component vector that contains boolean data.
GetBoolVectorArray	Get an array of four-component vectors that contain boolean data.
GetFloatVector	Get a four-component vector that contains floating-point data.
GetFloatVectorArray	Get an array of four-component vectors that contain floating-point data.
GetIntVector	Get a four-component vector that contains integer data.
GetIntVectorArray	Get an array of four-component vectors that contain integer data.
SetBoolVector	Set a four-component vector that contains boolean data.
SetBoolVectorArray	Set an array of four-component vectors that contain boolean data.
SetFloatVector	Set a four-component vector that contains floating-point data.
SetFloatVectorArray	Set an array of four-component vectors that contain floating-point data.
SetIntVector	Set a four-component vector that contains integer data.
SetIntVectorArray	Set an array of four-component vectors that contain integer data.

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVariable](#)

[Effects 11 Interfaces](#)

[D3DX Interfaces](#)

ID3DX11EffectVectorVariable::GetBoolVector method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get a four-component vector that contains boolean data.

Syntax

```
HRESULT GetBoolVector(  
    BOOL *pData  
)
```

Parameters

pData

Type: **BOOL***

A pointer to the first component.

Return value

Type: **HRESULT**

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVectorVariable](#)

ID3DX11EffectVectorVariable::GetBoolVectorArray method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get an array of four-component vectors that contain boolean data.

Syntax

```
HRESULT GetBoolVectorArray(  
    BOOL *pData,  
    UINT Offset,  
    UINT Count  
>;
```

Parameters

pData

Type: **BOOL***

A pointer to the start of the data to set.

Offset

Type: **UINT**

Must be set to 0; this is reserved for future use.

Count

Type: **UINT**

The number of array elements to set.

Return value

Type: **HRESULT**

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVectorVariable](#)

ID3DX11EffectVectorVariable::GetFloatVector method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Get a four-component vector that contains floating-point data.

Syntax

```
HRESULT GetFloatVector(  
    float *pData  
>;
```

Parameters

pData

Type: **float***

A pointer to the first component.

Return value

Type: **HRESULT**

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVectorVariable](#)

ID3DX11EffectVectorVariable::GetFloatVectorArray method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get an array of four-component vectors that contain floating-point data.

Syntax

```
HRESULT GetFloatVectorArray(  
    float *pData,  
    UINT  Offset,  
    UINT  Count  
>;
```

Parameters

pData

Type: **float***

A pointer to the start of the data to set.

Offset

Type: **UINT**

Must be set to 0; this is reserved for future use.

Count

Type: **UINT**

The number of array elements to set.

Return value

Type: **HRESULT**

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVectorVariable](#)

ID3DX11EffectVectorVariable::GetIntVector method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get a four-component vector that contains integer data.

Syntax

```
HRESULT GetIntVector(  
    int *pData  
)
```

Parameters

pData

Type: [int*](#)

A pointer to the first component.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVectorVariable](#)

ID3DX11EffectVectorVariable::GetIntVectorArray method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Get an array of four-component vectors that contain integer data.

Syntax

```
HRESULT GetIntVectorArray(  
    int *pData,  
    UINT Offset,  
    UINT Count  
>;
```

Parameters

pData

Type: [int*](#)

A pointer to the start of the data to set.

Offset

Type: [UINT](#)

Must be set to 0; this is reserved for future use.

Count

Type: [UINT](#)

The number of array elements to set.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVectorVariable](#)

ID3DX11EffectVectorVariable::SetBoolVector method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Set a four-component vector that contains boolean data.

Syntax

```
HRESULT SetBoolVector(  
    BOOL *pData  
)
```

Parameters

pData

Type: **BOOL***

A pointer to the first component.

Return value

Type: **HRESULT**

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVectorVariable](#)

ID3DX11EffectVectorVariable::SetBoolVectorArray method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Set an array of four-component vectors that contain boolean data.

Syntax

```
HRESULT SetBoolVectorArray(  
    BOOL *pData,  
    UINT Offset,  
    UINT Count  
>;
```

Parameters

pData

Type: [BOOL](#)*

A pointer to the start of the data to set.

Offset

Type: [UINT](#)

Must be set to 0; this is reserved for future use.

Count

Type: [UINT](#)

The number of array elements to set.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVectorVariable](#)

ID3DX11EffectVectorVariable::SetFloatVector method

2/22/2020 • 2 minutes to read • [Edit Online](#)

Set a four-component vector that contains floating-point data.

Syntax

```
HRESULT SetFloatVector(  
    float *pData  
)
```

Parameters

pData

Type: **float***

A pointer to the first component.

Return value

Type: **HRESULT**

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVectorVariable](#)

ID3DX11EffectVectorVariable::SetFloatVectorArray method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Set an array of four-component vectors that contain floating-point data.

Syntax

```
HRESULT SetFloatVectorArray(  
    float *pData,  
    UINT  Offset,  
    UINT  Count  
>;
```

Parameters

pData

Type: **float***

A pointer to the start of the data to set.

Offset

Type: **UINT**

Must be set to 0; this is reserved for future use.

Count

Type: **UINT**

The number of array elements to set.

Return value

Type: **HRESULT**

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVectorVariable](#)

ID3DX11EffectVectorVariable::SetIntVector method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Set a four-component vector that contains integer data.

Syntax

```
HRESULT SetIntVector(  
    int *pData  
)
```

Parameters

pData

Type: [int*](#)

A pointer to the first component.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVectorVariable](#)

ID3DX11EffectVectorVariable::SetIntVectorArray method

11/2/2020 • 2 minutes to read • [Edit Online](#)

Set an array of four-component vectors that contain integer data.

Syntax

```
HRESULT SetIntVectorArray(  
    int *pData,  
    UINT Offset,  
    UINT Count  
>;
```

Parameters

pData

Type: [int*](#)

A pointer to the start of the data to set.

Offset

Type: [UINT](#)

Must be set to 0; this is reserved for future use.

Count

Type: [UINT](#)

The number of array elements to set.

Return value

Type: [HRESULT](#)

Returns one of the following [Direct3D 11 Return Codes](#).

Remarks

NOTE

The DirectX SDK does not supply any compiled binaries for effects. You must use Effects 11 source to build your effects-type application. For more information about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	D3dx11effect.h
Library	N/A (An Effects 11 library is available online as shared source.)

See also

[ID3DX11EffectVectorVariable](#)

Effects 11 Functions

2/4/2021 • 2 minutes to read • [Edit Online](#)

This section contains information about the Effects 11 functions.

In this section

TOPIC	DESCRIPTION
D3DX11CreateEffectFromMemory	Creates an effect from a binary effect or file.

Related topics

[Effects 11 Reference](#)

D3DX11CreateEffectFromMemory function

11/2/2020 • 2 minutes to read • [Edit Online](#)

Creates an effect from a binary effect or file.

Syntax

```
HRESULT D3DX11CreateEffectFromMemory(
    void          *pData,
    SIZE_T        DataLength,
    UINT          FXFlags,
    ID3D11Device *pDevice,
    ID3DX11Effect **ppEffect
);
```

Parameters

pData

Type: **void***

Blob of compiled effect data.

DataLength

Type: **SIZE_T**

Length of the data blob.

FXFlags

Type: **UINT**

No effect flags exist. Set to zero.

pDevice

Type: **ID3D11Device***

Pointer to the **ID3D11Device** on which to create Effect resources.

ppEffect

Type: **ID3DX11Effect****

Address of the newly created **ID3DX11Effect** interface.

Return value

Type: **HRESULT**

The return value is one of the values listed in [Direct3D 11 Return Codes](#).

Remarks

NOTE

You must use [Effects 11 source](#) to build your effects-type application. For more info about using Effects 11 source, see [Differences Between Effects 10 and Effects 11](#).

Requirements

Header	
	D3dx11effect.h

See also

[Effects 11 Functions](#)

Effects 11 Structures

2/4/2021 • 2 minutes to read • [Edit Online](#)

This section contains information about the Effects 11 structures.

In this section

TOPIC	DESCRIPTION
D3DX11_EFFECT_DESC	Describes an effect.
D3DX11_EFFECT_SHADER_DESC	Describes an effect shader.
D3DX11_EFFECT_TYPE_DESC	Describes an effect-variable type.
D3DX11_EFFECT_VARIABLE_DESC	Describes an effect variable.
D3DX11_GROUP_DESC	Describes an effect group.
D3DX11_PASS_DESC	Describes an effect pass, which contains pipeline state.
D3DX11_PASS_SHADER_DESC	Describes an effect pass.
D3DX11_STATE_BLOCK_MASK	Indicates the device state.
D3DX11_TECHNIQUE_DESC	Describes an effect technique.

Related topics

[Effects 11 Reference](#)

D3DX11_EFFECT_DESC structure

11/2/2020 • 2 minutes to read • [Edit Online](#)

Describes an effect.

Syntax

```
typedef struct _D3DX11_EFFECT_DESC {  
    UINT ConstantBuffers;  
    UINT GlobalVariables;  
    UINT InterfaceVariables;  
    UINT Techniques;  
    UINT Groups;  
} D3DX11_EFFECT_DESC;
```

Members

ConstantBuffers

Type: [UINT](#)

Number of constant buffers in this effect.

GlobalVariables

Type: [UINT](#)

Number of global variables in this effect.

InterfaceVariables

Type: [UINT](#)

Number of global interfaces in this effect.

Techniques

Type: [UINT](#)

Number of techniques in this effect.

Groups

Type: [UINT](#)

Number of groups in this effect.

Remarks

D3DX11_EFFECT_DESC is used with [ID3DX11Effect::GetDesc](#).

Requirements

Header	D3dx11effect.h
--------	----------------

See also

[Effects 11 Structures](#)

D3DX11_EFFECT_SHADER_DESC structure

11/2/2020 • 2 minutes to read • [Edit Online](#)

Describes an effect shader.

Syntax

```
typedef struct _D3DX11_EFFECT_SHADER_DESC {
    const BYTE *pInputSignature;
    BOOL IsInline;
    const BYTE *pBytecode;
    UINT BytecodeLength;
    LPCSTR SODecls[D3D11_SO_STREAM_COUNT];
    UINT RasterizedStream;
    UINT NumInputSignatureEntries;
    UINT NumOutputSignatureEntries;
    UINT NumPatchConstantSignatureEntries;
} D3DX11_EFFECT_SHADER_DESC;
```

Members

pInputSignature

Type: **const BYTE***

Passed into `CreateInputLayout`. Only valid on a vertex shader or geometry shader. See [ID3D11Device::CreateInputLayout](#).

IsInline

Type: **BOOL**

TRUE if the shader is defined inline; otherwise FALSE.

pBytecode

Type: **const BYTE***

Shader bytecode.

BytecodeLength

Type: **UINT**

The length of pBytecode.

SODecls

Type: **LPCSTR**

Stream out declaration string (for geometry shader with SO).

RasterizedStream

Type: **UINT**

Indicates which stream is rasterized. D3D11 geometry shaders can output up to four streams of data, one of

which can be rasterized.

NumInputSignatureEntries

Type: **UINT**

Number of entries in the input signature.

NumOutputSignatureEntries

Type: **UINT**

Number of entries in the output signature.

NumPatchConstantSignatureEntries

Type: **UINT**

Number of entries in the patch constant signature.

Remarks

D3DX11_EFFECT_SHADER_DESC is used with [ID3DX11EffectShaderVariable::GetShaderDesc](#).

Requirements

Header	D3dx11effect.h
--------	----------------

See also

[Effects 11 Structures](#)

D3DX11_EFFECT_TYPE_DESC structure

11/2/2020 • 2 minutes to read • [Edit Online](#)

Describes an effect-variable type.

Syntax

```
typedef struct _D3DX11_EFFECT_TYPE_DESC {
    LPCSTR             TypeName;
    D3D10_SHADER_VARIABLE_CLASS Class;
    D3D10_SHADER_VARIABLE_TYPE   Type;
    UINT                Elements;
    UINT                Members;
    UINT                Rows;
    UINT                Columns;
    UINT                PackedSize;
    UINT                UnpackedSize;
    UINT                Stride;
} D3DX11_EFFECT_TYPE_DESC;
```

Members

TypeName

Type: [LPCSTR](#)

Name of the type, for example "float4" or "MyStruct".

Class

Type: [D3D10_SHADER_VARIABLE_CLASS](#)

The variable class (see [D3D10_SHADER_VARIABLE_CLASS](#)).

Type

Type: [D3D10_SHADER_VARIABLE_TYPE](#)

The variable type (see [D3D10_SHADER_VARIABLE_TYPE](#)).

Elements

Type: [UINT](#)

Number of elements in this type (0 if not an array).

Members

Type: [UINT](#)

Number of members (0 if not a structure).

Rows

Type: [UINT](#)

Number of rows in this type (0 if not a numeric primitive).

Columns

Type: **UINT**

Number of columns in this type (0 if not a numeric primitive).

PackedSize

Type: **UINT**

Number of bytes required to represent this data type, when tightly packed.

UnpackedSize

Type: **UINT**

Number of bytes occupied by this data type, when laid out in a constant buffer.

Stride

Type: **UINT**

Number of bytes to seek between elements, when laid out in a constant buffer.

Remarks

D3DX11_EFFECT_TYPE_DESC is used with [ID3DX11EffectType::GetDesc](#)

Requirements

Header	
	D3dx11effect.h

See also

[Effects 11 Structures](#)

D3DX11_EFFECT_VARIABLE_DESC structure

11/2/2020 • 2 minutes to read • [Edit Online](#)

Describes an effect variable.

Syntax

```
typedef struct _D3DX11_EFFECT_VARIABLE_DESC {  
    LPCSTR Name;  
    LPCSTR Semantic;  
    UINT    Flags;  
    UINT    Annotations;  
    UINT    BufferOffset;  
    UINT    ExplicitBindPoint;  
} D3DX11_EFFECT_VARIABLE_DESC;
```

Members

Name

Type: [LPCSTR](#)

Name of this variable, annotation, or structure member.

Semantic

Type: [LPCSTR](#)

Semantic string of this variable or structure member (NULL for annotations or if not present).

Flags

Type: [UINT](#)

Optional flags for effect variables.

Annotations

Type: [UINT](#)

Number of annotations on this variable (always 0 for annotations).

BufferOffset

Type: [UINT](#)

Offset into containing cbuffer or tbuffer (always 0 for annotations or variables not in constant buffers).

ExplicitBindPoint

Type: [UINT](#)

Used if the variable has been explicitly bound using the register keyword. Check Flags for D3DX11_EFFECT_VARIABLE_EXPLICIT_BIND_POINT.

Remarks

D3DX11_EFFECT_VARIABLE_DESC is used with [ID3DX11EffectVariable::GetDesc](#).

Requirements

Header	D3dx11effect.h
--------	----------------

See also

[Effects 11 Structures](#)

D3DX11_GROUP_DESC structure

11/2/2020 • 2 minutes to read • [Edit Online](#)

Describes an effect group.

Syntax

```
typedef struct _D3DX11_GROUP_DESC {  
    LPCSTR Name;  
    UINT    Techniques;  
    UINT    Annotations;  
} D3DX11_GROUP_DESC;
```

Members

Name

Type: [LPCSTR](#)

Name of this group (only **NULL** if global).

Techniques

Type: [UINT](#)

Number of techniques contained in group.

Annotations

Type: [UINT](#)

Number of annotations on this group.

Remarks

`D3DX11_GROUP_DESC` is used with [ID3DX11EffectTechnique::GetDesc](#).

Requirements

Header	D3dx11effect.h

See also

[Effects 11 Structures](#)

D3DX11_PASS_DESC structure

11/2/2020 • 2 minutes to read • [Edit Online](#)

Describes an effect pass, which contains pipeline state.

Syntax

```
typedef struct _D3DX11_PASS_DESC {  
    LPCSTR Name;  
    UINT Annotations;  
    BYTE *pIAInputSignature;  
    SIZE_T IAInputSignatureSize;  
    UINT StencilRef;  
    UINT SampleMask;  
    FLOAT BlendFactor[4];  
} D3DX11_PASS_DESC;
```

Members

Name

Type: [LPCSTR](#)

Name of this pass (NULL if not anonymous).

Annotations

Type: [UINT](#)

Number of annotations on this pass.

pIAInputSignature

Type: [BYTE*](#)

Signature from the vertex shader or geometry shader (if there is no vertex shader) or NULL if neither exists.

IAInputSignatureSize

Type: [SIZE_T](#)

Singature size in bytes.

StencilRef

Type: [UINT](#)

The stencil-reference value used in the depth-stencil state.

SampleMask

Type: [UINT](#)

The sample mask for the blend state.

BlendFactor

Type: [FLOAT](#)

The per-component blend factors (RGBA) for the blend state.

Remarks

D3DX11_PASS_DESC is used with [ID3DX11EffectPass::GetDesc](#).

Requirements

Header	
	D3dx11effect.h

See also

[Effects 11 Structures](#)

D3DX11_PASS_SHADER_DESC structure

11/2/2020 • 2 minutes to read • [Edit Online](#)

Describes an effect pass.

Syntax

```
typedef struct _D3DX11_PASS_SHADER_DESC {  
    ID3DX11EffectShaderVariable *pShaderVariable;  
    UINT ShaderIndex;  
} D3DX11_PASS_SHADER_DESC;
```

Members

pShaderVariable

Type: [ID3DX11EffectShaderVariable*](#)

The variable that this shader came from.

ShaderIndex

Type: [UINT](#)

The element of pShaderVariable (if an array) or 0 if not applicable.

Remarks

D3DX11_PASS_SHADER_DESC is used with [ID3DX11EffectPass](#) Get*ShaderDesc methods.

If this is an inline shader assignment, the returned interface will be an anonymous shader variable, which is not retrievable any other way. Its name in the variable description will be "\$Anonymous". If there is no assignment of this type in the pass block, pShaderVariable != **NULL**, but pShaderVariable->IsValid() == **FALSE**.

Requirements

Header	
	D3dx11effect.h

See also

[Effects 11 Structures](#)

D3DX11_STATE_BLOCK_MASK structure

11/2/2020 • 4 minutes to read • [Edit Online](#)

Indicates the device state.

Syntax

```
typedef struct _D3DX11_STATE_BLOCK_MASK {
    BYTE VS;
    BYTE VSSamplers[D3DX11_BYTES_FROM_BITS(D3D11_COMMONSHADER_SAMPLER_SLOT_COUNT)];
    BYTE VSShaderResources[D3DX11_BYTES_FROM_BITS(D3D11_COMMONSHADER_INPUT_RESOURCE_SLOT_COUNT)];
    BYTE VSConstantBuffers[D3DX11_BYTES_FROM_BITS(D3D11_COMMONSHADER_CONSTANT_BUFFER_API_SLOT_COUNT)];
    BYTE VSInterfaces[D3DX11_BYTES_FROM_BITS(D3D11_SHADER_MAX_INTERFACES)];
    BYTE HS;
    BYTE HSSamplers[D3DX11_BYTES_FROM_BITS(D3D11_COMMONSHADER_SAMPLER_SLOT_COUNT)];
    BYTE HSShaderResources[D3DX11_BYTES_FROM_BITS(D3D11_COMMONSHADER_INPUT_RESOURCE_SLOT_COUNT)];
    BYTE HSConstantBuffers[D3DX11_BYTES_FROM_BITS(D3D11_COMMONSHADER_CONSTANT_BUFFER_API_SLOT_COUNT)];
    BYTE HSInterfaces[D3DX11_BYTES_FROM_BITS(D3D11_SHADER_MAX_INTERFACES)];
    BYTE DS;
    BYTE DSSamplers[D3DX11_BYTES_FROM_BITS(D3D11_COMMONSHADER_SAMPLER_SLOT_COUNT)];
    BYTE DSShaderResources[D3DX11_BYTES_FROM_BITS(D3D11_COMMONSHADER_INPUT_RESOURCE_SLOT_COUNT)];
    BYTE DSConstantBuffers[D3DX11_BYTES_FROM_BITS(D3D11_COMMONSHADER_CONSTANT_BUFFER_API_SLOT_COUNT)];
    BYTE DSInterfaces[D3DX11_BYTES_FROM_BITS(D3D11_SHADER_MAX_INTERFACES)];
    BYTE GS;
    BYTE GSSamplers[D3DX11_BYTES_FROM_BITS(D3D11_COMMONSHADER_SAMPLER_SLOT_COUNT)];
    BYTE GSShaderResources[D3DX11_BYTES_FROM_BITS(D3D11_COMMONSHADER_INPUT_RESOURCE_SLOT_COUNT)];
    BYTE GSConstantBuffers[D3DX11_BYTES_FROM_BITS(D3D11_COMMONSHADER_CONSTANT_BUFFER_API_SLOT_COUNT)];
    BYTE GSInterfaces[D3DX11_BYTES_FROM_BITS(D3D11_SHADER_MAX_INTERFACES)];
    BYTE PS;
    BYTE PSSamplers[D3DX11_BYTES_FROM_BITS(D3D11_COMMONSHADER_SAMPLER_SLOT_COUNT)];
    BYTE PSShaderResources[D3DX11_BYTES_FROM_BITS(D3D11_COMMONSHADER_INPUT_RESOURCE_SLOT_COUNT)];
    BYTE PSConstantBuffers[D3DX11_BYTES_FROM_BITS(D3D11_COMMONSHADER_CONSTANT_BUFFER_API_SLOT_COUNT)];
    BYTE PSInterfaces[D3DX11_BYTES_FROM_BITS(D3D11_SHADER_MAX_INTERFACES)];
    BYTE PSUnorderedAccessViews;
    BYTE CS;
    BYTE CSSamplers[D3DX11_BYTES_FROM_BITS(D3D11_COMMONSHADER_SAMPLER_SLOT_COUNT)];
    BYTE CSShaderResources[D3DX11_BYTES_FROM_BITS(D3D11_COMMONSHADER_INPUT_RESOURCE_SLOT_COUNT)];
    BYTE CSConstantBuffers[D3DX11_BYTES_FROM_BITS(D3D11_COMMONSHADER_CONSTANT_BUFFER_API_SLOT_COUNT)];
    BYTE CSIInterfaces[D3DX11_BYTES_FROM_BITS(D3D11_SHADER_MAX_INTERFACES)];
    BYTE CSUnorderedAccessViews;
    BYTE IAVertexBuffers[D3DX11_BYTES_FROM_BITS(D3D11_IA_VERTEX_INPUT_RESOURCE_SLOT_COUNT)];
    BYTE IAIndexBuffer;
    BYTE IAInputLayout;
    BYTE IAPrimitiveTopology;
    BYTE OMRenderTargets;
    BYTE OMDepthStencilState;
    BYTE OMBlendState;
    BYTE RSViewports;
    BYTE RSScissorRects;
    BYTE RSRasterizerState;
    BYTE SOBuffers;
    BYTE Predication;
} D3DX11_STATE_BLOCK_MASK;
```

Members

VS

Type: **BYTE**

Boolean value indicating whether to save the vertex shader state.

VSSamplers

Type: **BYTE**

Array of vertex-shader samplers. The array is a multi-byte bitmask where each bit represents one sampler slot.

VSShaderResources

Type: **BYTE**

Array of vertex-shader resources. The array is a multi-byte bitmask where each bit represents one resource slot.

VSConstantBuffers

Type: **BYTE**

Array of vertex-shader constant buffers. The array is a multi-byte bitmask where each bit represents one constant buffer slot.

VSInterfaces

Type: **BYTE**

Array of vertex-shader interfaces. The array is a multi-byte bitmask where each bit represents one interface slot.

HS

Type: **BYTE**

Boolean value indicating whether to save the hull shader state.

HSSamplers

Type: **BYTE**

Array of hull-shader samplers. The array is a multi-byte bitmask where each bit represents one sampler slot.

HSShaderResources

Type: **BYTE**

Array of hull-shader resources. The array is a multi-byte bitmask where each bit represents one resource slot.

HSConstantBuffers

Type: **BYTE**

Array of hull-shader constant buffers. The array is a multi-byte bitmask where each bit represents one constant buffer slot.

HSInterfaces

Type: **BYTE**

Array of hull-shader interfaces. The array is a multi-byte bitmask where each bit represents one interface slot.

DS

Type: **BYTE**

Boolean value indicating whether to save the domain shader state.

DSSamplers

Type: **BYTE**

Array of domain-shader samplers. The array is a multi-byte bitmask where each bit represents one sampler slot.

DSShaderResources

Type: **BYTE**

Array of domain-shader resources. The array is a multi-byte bitmask where each bit represents one resource slot.

DSConstantBuffers

Type: **BYTE**

Array of domain-shader constant buffers. The array is a multi-byte bitmask where each bit represents one buffer slot.

DSInterfaces

Type: **BYTE**

Array of domain-shader interfaces. The array is a multi-byte bitmask where each bit represents one interface slot.

GS

Type: **BYTE**

Boolean value indicating whether to save the geometry shader state.

GSSamplers

Type: **BYTE**

Array of geometry-shader samplers. The array is a multi-byte bitmask where each bit represents one sampler slot.

GSShaderResources

Type: **BYTE**

Array of geometry-shader resources. The array is a multi-byte bitmask where each bit represents one resource slot.

GSConstantBuffers

Type: **BYTE**

Array of geometry-shader constant buffers. The array is a multi-byte bitmask where each bit represents one buffer slot.

GSInterfaces

Type: **BYTE**

Array of geometry-shader interfaces. The array is a multi-byte bitmask where each bit represents one interface slot.

PS

Type: **BYTE**

Boolean value indicating whether to save the pixel shader state.

PSSamplers

Type: **BYTE**

Array of pixel-shader samplers. The array is a multi-byte bitmask where each bit represents one sampler slot.

PSShaderResources

Type: **BYTE**

Array of pixel-shader resources. The array is a multi-byte bitmask where each bit represents one resource slot.

PSConstantBuffers

Type: **BYTE**

Array of pixel-shader constant buffers. The array is a multi-byte bitmask where each bit represents one constant buffer slot.

PSInterfaces

Type: **BYTE**

Array of pixel-shader interfaces. The array is a multi-byte bitmask where each bit represents one interface slot.

PSUnorderedAccessViews

Type: **BYTE**

Boolean value indicating whether to save the pixel shader unordered access views.

CS

Type: **BYTE**

Boolean value indicating whether to save the compute shader state.

CSSamplers

Type: **BYTE**

Array of compute-shader samplers. The array is a multi-byte bitmask where each bit represents one sampler slot.

CSShaderResources

Type: **BYTE**

Array of compute-shader resources. The array is a multi-byte bitmask where each bit represents one resource slot.

CSConstantBuffers

Type: **BYTE**

Array of compute-shader constant buffers. The array is a multi-byte bitmask where each bit represents one constant buffer slot.

CSInterfaces

Type: **BYTE**

Array of compute-shader interfaces. The array is a multi-byte bitmask where each bit represents one interface

slot.

CSUnorderedAccessViews

Type: **BYTE**

Boolean value indicating whether to save the compute shader unordered access views.

IAVertexBuffers

Type: **BYTE**

Array of vertex buffers. The array is a multi-byte bitmask where each bit represents one resource slot.

IAIndexBuffer

Type: **BYTE**

Boolean value indicating whether to save the index buffer state.

IAInputLayout

Type: **BYTE**

Boolean value indicating whether to save the input layout state.

IAPrimitiveTopology

Type: **BYTE**

Boolean value indicating whether to save the primitive topology state.

OMRenderTargets

Type: **BYTE**

Boolean value indicating whether to save the render targets states.

OMDepthStencilState

Type: **BYTE**

Boolean value indicating whether to save the depth-stencil state.

OMBlendState

Type: **BYTE**

Boolean value indicating whether to save the blend state.

RSViewports

Type: **BYTE**

Boolean value indicating whether to save the viewports states.

RSScissorRects

Type: **BYTE**

Boolean value indicating whether to save the scissor rectangles states.

RSRasterizerState

Type: **BYTE**

Boolean value indicating whether to save the rasterizer state.

SOBuffers

Type: **BYTE**

Boolean value indicating whether to save the stream-out buffers states.

Predication

Type: **BYTE**

Boolean value indicating whether to save the predication state.

Remarks

A state-block mask indicates the device states that a pass or a technique changes.

Requirements

Header	D3dx11effect.h

See also

[Effects 11 Structures](#)

D3DX11_TECHNIQUE_DESC structure

11/2/2020 • 2 minutes to read • [Edit Online](#)

Describes an effect technique.

Syntax

```
typedef struct _D3DX11_TECHNIQUE_DESC {  
    LPCSTR Name;  
    UINT    Passes;  
    UINT    Annotations;  
} D3DX11_TECHNIQUE_DESC;
```

Members

Name

Type: [LPCSTR](#)

Name of this technique (NULL if not anonymous).

Passes

Type: [UINT](#)

Number of passes contained in the technique.

Annotations

Type: [UINT](#)

Number of annotations on this technique.

Remarks

`D3DX11_TECHNIQUE_DESC` is used with [ID3DX11EffectTechnique::GetDesc](#).

Requirements

Header	D3dx11effect.h

See also

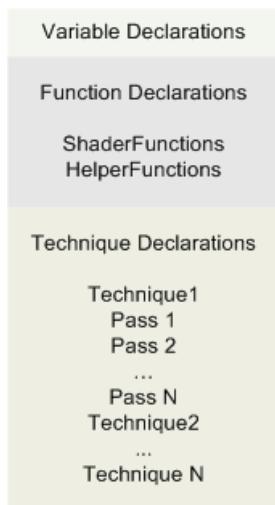
[Effects 11 Structures](#)

Effect Format (Direct3D 11)

2/4/2021 • 2 minutes to read • [Edit Online](#)

An effect (which is often stored in a file with a .fx file extension) declares the pipeline state set by an effect. Effect state can be roughly broken down into three categories:

- [Variables](#), which are usually declared at the top of an effect.
- [Functions](#), which implement shader code, or are used as helper functions by other functions.
- [Techniques](#), which can be arranged in effect groups, and implement rendering sequences using one or more effect passes. Each pass sets one or more [state groups](#) and calls shader functions.



The preceding diagram shows the categories of effect state.

The definition of the effect binary format can be found in Binary\EffectBinaryFormat.h in the effects source code.

In this section

TOPIC	DESCRIPTION
Effect Variable Syntax	An effect variable is declared with the syntax described in this section.
Annotation Syntax	An annotation is a user-defined piece of information, declared with the syntax described in this section.
Effect Function Syntax	An effect function is written in HLSL and is declared with the syntax described in this section.
Effect Technique Syntax	An effect technique is declared with the syntax described in this section.
Effect State Groups	Effect states are name value pairs in the form of an expression.
Effect Group Syntax	An effect group is declared with the syntax described in this section.

Related topics

[Effects 11 Reference](#)

Effect Variable Syntax (Direct3D 11)

11/2/2020 • 2 minutes to read • [Edit Online](#)

An effect variable is declared with the syntax described in this section.

Syntax

Basic syntax:

DataType *VariableName* [: *SemanticName*] < *Annotations* > [= *InitialValue*];

See [Variable Syntax \(DirectX HLSL\)](#) for full syntax.

NAME	DESCRIPTION
<i>DataType</i>	Any basic , texture , unordered access view, shader or state block type.
<i>VariableName</i>	An ASCII string that uniquely identifies the name of the effect variable.
<i>SemanticName</i>	A ASCII string that denotes additional information about how a variable should be used. A semantic is an ASCII string that can be either a predefined system-value or a custom-user string.
<i>Annotations</i>	One or more pieces of user-supplied information (metadata) that is ignored by the effect system. For syntax, see Annotation Syntax (Direct3D 11) .
<i>InitialValue</i>	The default value of the variable.

An effect variable that is declared outside of all functions, is considered global in scope; variables declared within a function are local to that function.

Example

This example illustrates global effect numeric variables.

```
float4 g_MaterialAmbientColor;      // Material's ambient color
float4 g_MaterialDiffuseColor;      // Material's diffuse color
float3 g_LightDir[3];               // Light's direction in world space
float4x4 g_mWorld;                 // World matrix for object
```

This example illustrates effect variables that are local to a shader function.

```
VS_OUTPUT RenderSceneVS( ... )
{
    float3 vNormalWorldSpace;
    float4 vAnimatedPos;

    // shader body
}
```

This example illustrates function parameters that have semantics.

```
VS_OUTPUT RenderSceneVS( float4 vPos : SV_POSITION,
                        float3 vNormal : NORMAL,
                        float2 vTexCoord0 : TEXCOORD0,
                        uniform int nNumLights,
                        uniform bool bTexture,
                        uniform bool bAnimate )

{
    ...
}
```

This example illustrates declaring a global texture variable.

```
Texture2D g_MeshTexture;           // Color texture for mesh
```

Sampling a texture is done with a texture sampler. To set up a sampler in an effect, see the [sampler type](#).

This example illustrates declaring global unordered access view variables.

```

RWStructuredBuffer<uint> bc : register(u2) < string name="bc"; >
RWBuffer<uint> bRW;
struct S
{
    uint key;
    uint value;
};
AppendStructuredBuffer<S> asb : register(u5);
RWByteAddressBuffer rwbab : register(u1);
RWStructuredBuffer<uint> rwsb : register(u3);
RWTexure1D<float> rwt1d : register(u1);
RWTexure1DArray<uint> rwt1da : register(u4);
RWTexure2D<uint> rwt2d : register(u2);
RWTexure2DArray<uint> rwt2da : register(u6);
RWTexure3D<uint> rwt3d : register(u7);
This example illustrates declaring global shader variables.
VertexShader pVS = CompileShader( vs_5_0, VS() );
HullShader pHS = NULL;
DomainShader pDS = NULL;
GeometryShader pGS = ConstructGSWithSO( CompileShader( gs_5_0, VS() ),
                                         "0:Position.xy; 1:Position.zw; 2:Color.xy",
                                         "3:Texcoord.xyzw; 3:$SKIP.x;",
                                         NULL,
                                         NULL,
                                         1 );
PixelShader pPS = NULL;
ComputeShader pCS = NULL;
This example illustrates declaring global state block variables.
BlendState myBS[2] < bool IsValid = true; >
{
{
    BlendEnable[0] = false;
},
{
    BlendEnable[0] = true;
    SrcBlendAlpha[0] = Inv_Src_Alpha;
}
};

RasterizerState myRS
{
    FillMode = Solid;
    CullMode = NONE;
    MultisampleEnable = true;
    DepthClipEnable = false;
};

DepthStencilState myDS
{
    DepthEnable = false;
    DepthWriteMask = Zero;
    DepthFunc = Less;
};
sampler mySS[2] : register(s3)
{
{
    Filter = ANISOTROPIC;
    MaxAnisotropy = 3;
},
{
    Filter = ANISOTROPIC;
    MaxAnisotropy = 4;
}
};

```

Related topics

[Effect Format](#)

Annotation Syntax (Direct3D 11)

11/2/2020 • 2 minutes to read • [Edit Online](#)

An annotation is a user-defined piece of information, declared with the syntax described in this section.

```
<DATATYPE NAME = VALUE; ... ;>
```

Parameters

ITEM	DESCRIPTION
<i>DataType</i>	[in] The data type, which includes any scalar HLSL type as well as the string type .
<i>Name</i>	[in] An ASCII string, that represents the annotation name.
<i>Value</i>	[in] The initial value of the annotation.
...	[in] Additional annotations (name-value pairs).

Remarks

You may add more than one annotation within the angle brackets, each one separated by a semicolon. The effect framework APIs recognize annotations on global variables; all other annotations are ignored.

Example

Here are some examples.

```
int i <int blabla=27; string blacksheep="Hello There";>;  
  
int j <int bambam=30; string blacksheep="Goodbye There";> = 5 ;  
  
float y <float y=2.3;> = 2.3, z <float y=1.3;> = 1.3 ;  
  
half w <half GlobalW = 3.62;>;  
  
float4 main(float4 pos : SV_POSITION ) : SV_POSITION  
{  
    pos.y = pos.x > 0 ? pos.w * 1.3 : pos.z * .032;  
    for (int x = i; x < j ; x++)  
    {  
        pos.w = pos.w * pos.y + x + j - y * w;  
    }  
  
    return pos;  
}
```

Related topics

[Effect Format](#)

Effect Variable Syntax

Effect Function Syntax (Direct3D 11)

11/2/2020 • 2 minutes to read • [Edit Online](#)

An effect function is written in HLSL and is declared with the syntax described in this section.

Syntax

```
ReturnType FunctionName( [ ArgumentList ] )  
{  
    \[ *Statements* \]  
};
```

NAME	DESCRIPTION
ReturnType	Any HLSL type
FunctionName	An ASCII string that uniquely identifies the name of the shader function.
ArgumentList	One or more arguments, separated by commas (see Function Arguments (DirectX HLSL)).
Statements	One or more statements (see Statements (DirectX HLSL)) that make up the body of the function. If a function is defined without a body, it is considered to be a prototype; and must be redefined with a body before use.

An effect function may be a shader or it may simply be a function called by a shader. A function is uniquely identified by its name, the types of its parameters, and the target platform; therefore, functions can be overloaded. Any valid HLSL function should fit this format; for a more detailed list of syntax for HLSL functions, see [Functions \(DirectX HLSL\)](#).

Example

The following is an example of a pixel shader function.

```
PS_OUTPUT RenderScenePS( VS_OUTPUT In,
                        uniform bool bTexture )
{
    PS_OUTPUT Output;

    // Lookup mesh texture and modulate it with diffuse
    if( bTexture )
        Output.RGBColor = g_MeshTexture.Sample(MeshTextureSampler, In.TextureUV) *
                          In.Diffuse;
    else
        Output.RGBColor = In.Diffuse;

    return Output;
}
```

Related topics

[Effect Format](#)

Effect Technique Syntax (Direct3D 11)

2/22/2020 • 2 minutes to read • [Edit Online](#)

An effect technique is declared with the syntax described in this section.

TechniqueVersion *TechniqueName* [<Annotations>]

{

```
pass *PassName* \[ <*annotations*> \] {  
  \[ *SetStateGroup*; \] \[ *SetStateGroup*; \] ... \[ *SetStateGroup*; \]  
}  
}
```

Parameters

ITEM	DESCRIPTION		
TechniqueVersion	Either technique10 or technique11. Techniques which use functionality new to Direct3D 11 (5_0 shaders, BindInterfaces, etc) must use technique11.		
<i>TechniqueName</i>	Optional. An ASCII string that uniquely identifies the name of the effect technique.		
<i>Annotations</i> >	[in] Optional. One or more pieces of user-supplied information (metadata) that is ignored by the effect system. For syntax, see Annotation Syntax (Direct3D 11) .		
pass	Required keyword.		
<i>PassName</i>	[in] Optional. An ASCII string that uniquely identifies the name of the pass.		
<i>SetStateGroup</i>	[in] Set one or more state groups such as: <table><thead><tr><th>STATEGROUP</th><th>SYNTAX</th></tr></thead></table>	STATEGROUP	SYNTAX
STATEGROUP	SYNTAX		

ITEM	DESCRIPTION	SYNTAX
	<p>Blend State</p> <p>See [ID3D11DeviceContext::OMSetBlendState] (/windows/desktop/api/D3D11/nf-d3d11-id3d11devicecontext-omsetblendstate) for the argument list.</p>	<pre>SetBlendState(arguments);</pre>
	<p>Depth-stencil State</p> <p>See ID3D11DeviceContext::OMSetDepthStencilState for the argument list.</p>	<pre>SetDepthStencilState(arguments);</pre>
	<p>Rasterizer State</p> <p>See [ID3D11DeviceContext::RSSetState] (/windows/desktop/api/D3D11/nf-d3d11-id3d11devicecontext-rssetstate) for the argument list.</p>	<pre>SetRasterizerState(arguments);</pre>
	<p>Shader State</p> <p>SetXXXShader is one of SetVertexShader, SetDomainShader, SetHullShader, SetGeometryShader, SetPixelShader or SetComputeShader (which are similar to the API methods ID3D11DeviceContext::VSSetShader, ID3D11DeviceContext::DSSetShader,</p>	<pre>SetXXXShader(Shader);</pre>

ITEM	DESCRIPTION	
	<p>ID3D11DeviceContext::HSSetShader, ID3D11DeviceContext::GSSetShader, ID3D11DeviceContext::PSSetShader and ID3D11DeviceContext::CSSetShader).</p> <p>Shader is a shader variable, which can be obtained in many ways:</p> <pre data-bbox="1144 467 1377 1073"> SetXXXShader(CompileShader(shader_profile, ShaderFunction(args))); SetXXXShader(CompileShader(NULL)); SetXXXShader(NULL); SetXXXShader(myShaderVar); SetXXXShader(myShaderArray[2]); SetXXXShader(myShaderArray[uIndex]); SetGeometryShader(ConstructGSSWithSO(Shader, strStream0)); SetGeometryShader(ConstructGSSWithSO(Shader, strStream0, strStream1, strStream2, strStream3, RastStream)); </pre>	
Render Target State	<p>One of:</p> <pre data-bbox="1144 1635 1377 1882"> SetRenderTarget(RTV0, DSV); SetRenderTarget(RTV0, RTV1, DSV); ... SetRenderTarget(RTV0, RTV1, RTV2, RTV3, RTV4, RTV5, RTV6, RTV7, DSV); </pre> <p>Similar to ID3D11DeviceContext::OMSetRenderTarget.</p>	

Examples

This example sets blending state.

```
BlendState NoBlend
{
    BlendEnable[0] = False;
};

...
technique10
{
    pass p2
    {
        ...
        SetBlendState( NoBlend, float4( 0.0f, 0.0f, 0.0f, 0.0f ), 0xFFFFFFFF );
    }
}
```

This example sets up the rasterizer state to render an object in wireframe.

```
RasterizerState rsWireframe { FillMode = WireFrame; };

...
technique10
{
    pass p1
    {
        ...
        SetRasterizerState( rsWireframe );
    }
}
```

This example sets shader state.

```
technique10 RenderSceneWithTexture1Light
{
    pass P0
    {
        SetVertexShader( CompileShader( vs_4_0, RenderSceneVS( 1, true, true ) ) );
        SetGeometryShader( NULL );
        SetPixelShader( CompileShader( ps_4_0, RenderScenePS( true ) ) );
    }
}
```

Related topics

[Effect Format](#)

[Effect Group Syntax \(Direct3D 11\)](#)

[Effect State Groups](#)

Effect State Groups (Direct3D 11)

11/2/2020 • 2 minutes to read • [Edit Online](#)

Effect states are name value pairs in the form of an expression.

- [Blend State](#)
- [Depth and Stencil State](#)
- [Rasterizer State](#)
- [Sampler State](#)
- [Effect Object State](#)
- [Defining and using state objects](#)
- [Related topics](#)

Blend State

ALPHACOVERAGEENABLEBLENDENABLESRCBLENDDESTBLENDOPSRCBLENDALPHADESTBLENDALPHABLENDOPALPHARENDRTARGETWRITEMASK	Members of D3D11_BLEND_DESC
--	---

Depth and Stencil State

DEPTHENABLEDEPTHWRITEMASKDEPTHFUNCSTENCILENABLESTENCILREADMASKSTENCILWRITEMASK	Members of D3D11_DEPTH_STENCIL_DESC
FRONTFACESTENCILFAILFRONTFACESTENCILZFAILFRONTFA CESTENCILPASSFRONTFACESTENCILFUNCBACKFACESTENCI LFAILBACKFACESTENCILZFAILBACKFACESTENCILPASSBACKF ACESTENCILFUNC	Member of D3D11_DEPTH_STENCILOP_DESC

Rasterizer State

FILLMODE	D3D11_FILL_MODE
CULLMODE	D3D11_CULL_MODE
FRONTCOUNTERCLOCKWISEDEPTHBIASDEPTHBIASCLAMPS LOPESCALEDDDEPTHBIAS ZCLIPENABLESCISSORENABLEMULTISAMPLEENABLEANTIAL IASEDLINEENABLE	Members of D3D11_RASTERIZER_DESC

Sampler State

Filter AddressU AddressV AddressW MipLODBias MaxAnisotropy ComparisonFunc BorderColor MinLOD MaxLOD	Members of D3D11_SAMPLER_DESC
---	---

See [Sampler Type \(DirectX HLSL\)](#) for examples.

Effect Object State

THIS EFFECT OBJECT	MAPS TO
RASTERIZERSTATE	A Rasterizer State state object.
DEPTHSTENCILSTATE	A Depth and Stencil State state object.
BLENDSTATE	A Blend State state object.
VERTEXSHADER	A compiled vertex shader object.
PIXELSHADER	A compiled pixel shader object.
GEOMETRYSHADER	A compiled geometry shader object.
DS_STENCILREFAB_BLENDFACTORAB_SAMPLEMASK	Members of D3DX11_PASS_DESC .

Defining and using state objects

State objects are declared in FX files in the following format. StateObjectType is one of the states listed above and MemberName is the name of any member that will have a non-default value.

```
StateObjectType ObjectType {
    MemberName = value;
    ...
    MemberName = value;
};
```

For example, to set up a blend state object with AlphaToCoverageEnable and BlendEnable[0] set to FALSE the following code would be used.

```
BlendState NoBlend {
    AlphaToCoverageEnable = FALSE;
    BlendEnable[0] = FALSE;
};
```

The state object is applied to a technique pass using one of the SetStateGroup functions described in [Effect Technique Syntax \(Direct3D 11\)](#). For example, to apply the BlendState object described above the following code

would be used.

```
SetBlendState( NoBlend, float4( 0.0f, 0.0f, 0.0f, 0.0f ), 0xFFFFFFFF );
```

Related topics

[Effect Technique Syntax](#)

[Effect Format \(Direct3D 11\)](#)

Effect Group Syntax (Direct3D 11)

2/22/2020 • 2 minutes to read • [Edit Online](#)

An effect group is declared with the syntax described in this section.

```
fxgroup GroupName [ <Annotations> ]
{
    TechniqueVersion TechniqueName [ <Annotations> ]
    {
        ...
    }
    TechniqueVersion TechniqueName [ <Annotations> ]
    {
        ...
    }
}
```

Parameters

ITEM	DESCRIPTION
fxgroup	Required keyword.
GroupName	Required. An ASCII string that uniquely identifies the name of the effect group. Unlike techniques, groups must have names to ensure that techniques have a unique identifier (see Groups and Techniques section below).
< Annotations >	[in] Optional. One or more pieces of user-supplied information (metadata) that is ignored by the effect system. For syntax, see Annotation Syntax (Direct3D 11).
TechniqueVersion	Either "technique10" or "technique11". Techniques which use functionality new to Direct3D 11 (5_0 shaders, BindInterfaces, etc) must use "technique11".
TechniqueName	Optional. An ASCII string that uniquely identifies the name of the effect technique.

Groups and Techniques

In order to maintain compatibility with fx_4_0 effects, groups are optional. There is an implicit NULL-named group surrounding all global techniques.

Consider the following example:

```
technique11 GlobalTech
{
}
fxgroup Group1
{
    technique11 Tech1 { ... }
    technique11 Tech2 { ... }
}
fxgroup Group2
{
    technique11 Tech1 { ... }
    technique11 Tech2 { ... }
}
```

In C++, one can get a technique by name in two ways. The following commands will find the obvious techniques:

```
pEffect->GetTechniqueByName( "GlobalTech" );
pEffect->GetTechniqueByName( "|GlobalTech" );
pEffect->GetTechniqueByName( "Group1|Tech1" );
pEffect->GetTechniqueByName( "Group1|Tech2" );
pEffect->GetTechniqueByName( "Group2|Tech1" );
pEffect->GetTechniqueByName( "Group2|Tech2" );
pEffect->GetGroupByName("Group1")->GetTechniqueByName( "Tech1" );
pEffect->GetGroupByName("Group1")->GetTechniqueByName( "Tech2" );
pEffect->GetGroupByName("Group2")->GetTechniqueByName( "Tech1" );
pEffect->GetGroupByName("Group2")->GetTechniqueByName( "Tech2" );
```

In order to ensure that [ID3DX11Effect::GetTechniqueByName](#) works similarly to Effects 10, all defined groups must have a name.

Related topics

[Effect Format](#)

[Effect Technique Syntax \(Direct3D 11\)](#)