Tell us about your PDF experience.

Direct3D 10 Graphics

Article • 01/06/2021 • 2 minutes to read

This section contains information about programming with Microsoft Direct3D 10.

In This Section

Item	Description
Programming Guide for Direct3D 10	This section contains architecture descriptions, functional block diagrams, descriptions of the building blocks in the pipeline, and code snippets.
Reference for Direct3D 10	This section contains the reference pages for the graphics components including the syntax for the API methods, functions, and data structures.

Programming Guide for Direct3D 10

Article • 01/06/2021 • 2 minutes to read

The programming guide contains information about how to use the Direct3D 10 programmable pipeline to create realtime 3D graphics for games as well as scientific and desktop applications.

- API Features
- Pipeline Stages
- Resources
- Effects
- Direct3D 9 to Direct3D 10 Considerations
- Linking Static and Dynamic Libraries
- Direct3D 10.1 Features

Related topics

Direct3D 10 Graphics

API Features (Direct3D 10)

Article • 01/06/2021 • 2 minutes to read

The Direct3D 10 graphics pipeline represents a fundamental architecture change, rebuilt from the ground-up in hardware and software to power the next-generation of games and 3D multimedia applications. It uses the Windows Display Driver Model (WDDM), which enables performance and behavioral enhancements such as virtual GPU memory.

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

- The ability to process entire primitives in the new geometry-shader stage.
- The ability to output pipeline-generated vertex data to memory using the stream-output stage.
- Organization of pipeline state into 5 immutable state objects, enabling fast configuration of the pipeline.
- Organization of shader constants into constant buffers, minimizing bandwidth overhead for supplying shader-constant data.
- The ability to perform per-primitive material swapping and setup using a geometry shader.
- New resource types (including texture arrays that can be indexed from shaders) and resource formats.
- Increased generalization of resource access using a view.
- Legacy hardware capability bits (caps) have been removed in favor of a rich set of guaranteed functionality, which targets Direct3D 10-class hardware (minimum).
- Layered Runtime - The Direct3D 10 API is constructed with layers, starting with the basic functionality at the core and building optional and developer-assist functionality (debug, etc.) in outer layers.
- Full HLSL integration - All Direct3D 10 shaders are written in HLSL and implemented with the common-shader core.
- An increase in the number of render targets, textures, and samplers. There is also no shader length limit.
- Integer and bitwise shader operations.
- Readback of a depth/stencil surface or a multisampled resource, once it is no longer bound as a render target.
- Multisampled alpha-to-coverage support.

There are additional behavioral differences that Direct3D 9 developers should also be aware of (see Direct3D 9 to Direct3D 10 Considerations).

Here is a list of Direct3D 9 features that either are no longer supported, or have been revised in Direct3D 10 (see Deprecated Features).

Related topics

Programming Guide for Direct3D 10

State Objects (Direct3D 10)

Article • 05/24/2021 • 2 minutes to read

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

- Input-Layout State
- Rasterizer State
- Depth-Stencil State
- Blend State
- Sampler State
- Performance Considerations
- Related topics

Input-Layout State

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

To create a input-layout-state object, see **CreateInputLayout**.

Rasterizer State

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

To create a rasterizer-state object, see **CreateRasterizerState**.

Depth-Stencil State

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

To create a depth-stencil-state object, see **CreateDepthStencilState**.

Blend State

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

To create a blend-state object, see **CreateBlendState**.

Sampler State

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

Differences between Direct3D 9 and Direct3D 10:

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

To create a sampler-state object, see **CreateSamplerState**.

Performance Considerations

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

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

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

Related topics

API Features (Direct3D 10)

API Layers (Direct3D 10)

Article • 05/24/2021 • 3 minutes to read

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

Core Layer

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

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

Create layers when a device is created by calling **D3D10CreateDevice** and supplying one or more **D3D10_CREATE_DEVICE_FLAG** values.

Debug Layer

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

To create a device that supports the debug layer, you must install the DirectX SDK (to get D3D10SDKLayers.DLL), and then specify the D3D10_CREATE_DEVICE_DEBUG flag when calling **D3D10CreateDevice**. Of course, running an application with the debug layer will be substantially slower. To enable/disable debug messages for D3DX10 APIs, call **D3DX10DebugMute**.

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 **ID3D10DeviceChild::SetPrivateData** method and the **WKPDID_D3DDebugObjectName** 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_D3DDebugObjectName, sizeof( c_szName ) - 1,
c_szName );
```

Typically, you should compile these calls out of your production version.

Switch-to-Reference Layer

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

All device states, resources and objects are maintained through this device transition. However, it is sometimes not possible to exactly match resource data. This is due to the fact that some information is available to a hardware device that is not available to a reference device.

Virtually all real-time applications use the HAL implementation of the pipeline. When the pipeline is switched from a hardware device to a reference device, rendering operations are done simultaneously in both hardware and software. As the software device is rendering, it will require that resources are downloaded to system memory. This may require other resources cached in system memory to be evicted to make room.

> **Note**
>
> This switch-to-reference-layer feature is supported only in Direct3D 10 and Direct3D 10.1, and is no longer supported in Direct3D 11 and later versions.

Thread-Safe Layer

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

A Direct3D 10 application can control a device synchronization using device functions. This enables an application to enable/disable critical sections (temporarily enabling/disabling multithreading protection), and take/release a critical-section lock

over multiple Direct3D 10 API entry points. The thread-safe layer is enabled by default. For single-threaded applications, the thread-safe layer has no impact on performance.

Differences between Direct3D 9 and Direct3D 10:

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

Related topics

API Features (Direct3D 10)

Customize Debug Output with ID3D10InfoQueue (Direct3D 10)

Article • 04/26/2022 • 3 minutes to read

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

In general:

- Call **ID3D10InfoQueue::AddApplicationMessage** to generate user-defined messages
- Call **ID3D10InfoQueue::GetMessage** is used to get messages (that pass an optional retrieval filter).

Registry Controls

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

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

Where:

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

Filtering Debug Messages using Registry Keys

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

- DWORD Mute_CATEGORY_* - Debug output if this key is non-zero.
- DWORD Mute_SEVERITY_* - Debug output is disabled if this key is non-zero.
- DWORD Mute_ID_* - Message name or number can be used for * (just like for BreakOn_ID_* described earlier). Debug output is disabled if this key is non-zero.
- DWORD Unmute_SEVERITY_INFO - Debug output is ENABLED if this key is non-zero. By default when InfoQueueStorageFilterOverride is enabled, debug messages with severity INFO are muted - therefore for this key allows INFO to be turned back on.

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

Setting Break Conditions using Registry Keys

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

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

- **BreakOn_CATEGORY_*** - Break on any message passing through the storage filters. * is one of the D3D10_MESSAGE_CATEGORY messages.
- **BreakOn_SEVERITY_*** - Break on any message passing through the storage filters. * is one of the D3D10_MESSAGE_SEVERITY_ messages.
- **BreakOn_ID_*** - Break on any message passing through the storage filters. * is one of the D3D10_MESSAGE_ID_ messages or can be the numerical value of the error enumeration. For example, Suppose the message with ID "D3D10_MESSAGE_ID_HYPOTHETICAL" had the value 123 in the D3D10_MESSAGE_ID enumeration. In this case, creating the value BreakOn_ID_HYPOTHETICAL=1 or BreakOn_ID_123=1 would both accomplish the same thing - break when a message having ID D3D10_MESSAGE_ID_HYPOTHETICAL is encountered.

Muting Debug Output using Registry Keys

Debug output can be muted using a MuteDebugOutput key. The presence of this value in the registry forces override of the InfoQueue's ID3D10InfoQueue::SetMuteDebugOutput method. MuteDebugOutput stops messages that pass the storage filter from being sent to debug output.

Disabling Debug Layer Messages

Debug layer messages can be disabled individualy or as a group at runtime by specifying filters using **ID3D10InfoQueue::AddStorageFilterEntries**. The *pFilter* argument to **ID3D10InfoQueue::AddStorageFilterEntries** takes a **D3D10_INFO_QUEUE_FILTER** structure that contains an allow list and a deny list. The allow and deny lists are described by **D3D10_INFO_QUEUE_FILTER_DESC** structures that allow filtering to be specified by catergory, severity and individual message ID.

The following code is an example of setting up the **ID3D10InfoQueue Interface** to deny the D3D10_MESSAGE_ID_DEVICE_DRAW_INDEX_BUFFER_TOO_SMALL message.

```
  //retrieve the ID3D10InfoQueue from a Direct3D device created with the
D3D10_CREATE_DEVICE_DEBUG flag
  ID3D10InfoQueue * pInfoQueue;
    g_pd3dDevice->QueryInterface( __uuidof(ID3D10InfoQueue),  (void
**)&pInfoQueue );

  //set up the list of messages to filter
    D3D10_MESSAGE_ID messageIDs [] = {
D3D10_MESSAGE_ID_DEVICE_DRAW_INDEX_BUFFER_TOO_SMALL };

  //set the DenyList to use the list of messages
    D3D10_INFO_QUEUE_FILTER filter = { 0 };
    filter.DenyList.NumIDs = 1;
    filter.DenyList.pIDList = messageIDs;

  //apply the filter to the info queue
    pInfoQueue->AddStorageFilterEntries( &filter );
```

Related topics

API Layers

API Features (Direct3D 10)

Reference Counting (Direct3D 10)

Article • 05/24/2021 • 2 minutes to read

Direct3D 10 set functions do not hold a reference to a device-child object. This means that each application must hold a reference to a device-child object for as long as the object needs to be bound to the pipeline. When the reference count of an object drops to zero, the object will be unbound from the pipeline and destroyed. This style of reference holding is also known as weak-reference holding because each pipeline binding location holds a weak reference to the interface/object that is bound to it.

For example:

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

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

pRasterizerState->Release();
// Following the final release, the object is unbound.
pDevice->RSGetState( &pCurRasterizerState );
// pCurRasterizerState will be equal to NULL.
```

Differences between Direct3D 9 and Direct3D 10:

- In Direct3D 9, set functions hold a reference to the device objects; in Direct3D 10 set functions do not hold a reference to the device-child objects.

Related topics

API Features (Direct3D 10)

Deprecated Features (Direct3D 10)

Article • 05/24/2021 • 3 minutes to read

A list of the features available in Direct3D 10 is here. This page lists the Direct3D 9 features that are no longer supported in Direct3D 10.

The biggest feature changes in Direct3D 10 are:

- Direct3D 10 no longer supports the fixed-function transform and lighting pipeline.
- Direct3D 10 no longer supports the fixed-function texture blender (sometimes called a fixed-function pixel shader).
- Direct3D 10 implements new rasterization rules, which are simpler and cleaner than the legacy GDI rules that are implemented in Direct3D 9. For instance, the last-pixel control for lines is no longer supported.

Here is a complete list of the features in Direct3D 9 that have been deprecated in Direct3D 10.

- **Alpha blend**. Alpha blend is now programmed independent of color blend. Direct3D 10 adds an alpha-blend-enable toggle which is enabled by default. See State Objects (Direct3D 10) for more information.

- **Alpha test**. Alpha test is a fixed-function pixel behavior for Direct3D 9. Alpha test is moved into programmable pixel shaders for Direct3D 10 and higher. For info about emulating the Direct3D 9 alpha test functionality in Direct3D 10 and higher, see the FixedFuncEMU sample in the DirectX SDK for June 2010.

- **Blend mode options**. BOTHSRCALPHA has been removed from D3D10_BLEND since it is redundant with BOTHINVSRCALPHA. See **D3D10_BLEND** for more information.

- **Block compression formats**. There is no distinction between pre-multiplied alpha or non-premultiplied alpha in Direct3D 10. These Direct3D 9 formats map to these Direct3D 10 formats:

Direct3D 9	Direct3D 10
DXT1	BC1*
DXT2,DXT3	BC2*
DXT4,DXT5	BC3*

See Block Compression (Direct3D 10) for additional information.

- **Clip planes**. Instead of using clip planes, Direct3D 10 implements clip distances and cull distances, with up to 8 components each in up to 2 elements of vertex attributes. See Semantics (DirectX HLSL) for additional information. FixedFuncEMU sample provides an example of emulating clip planes in Direct3D 10.

- **Dithering**. Direct3D 10 does not support writing dithered data to a render target.

- **Fixed-function transform and lighting pipeline in not available**. Instead, you must use shaders. See Shader Stages (Direct3D 10) for additional information.

- **Fixed-function texture blender (also called a fixed function pixel shader)**. Instead, you must use shaders. See Shader Stages (Direct3D 10) for additional information.

- **Fill modes** have changed. Direct3D 10 implements solid and wireframe fill modes. D3DFILLMODE point has been removed, use a geometry shader to emulate point mode if necessary. FixedFuncEMU sample provides an example of emulating D3DFILLMODE point in Direct3D 10. See **D3D10_FILL_MODE** and Shader Stages (Direct3D 10) for additional information.

- **Formats**. Hardware can use formats exposed by the API. Luminance formats are no longer implemented.

- **Mipmap filtering**. Removed the option for selecting no-filter mode. Instead, use a texture with only a single mipmap or set the MaxLOD sampler state to 0. See State Objects (Direct3D 10) for additional information.

- **Palettes**. Applications should use a dependent texture read instead.

- **Pixel and vertex shader models**: 1_x, 2_x, and 3_0. Direct3D 10 supports shader model 4. See Shader Model 4 for additional information.

- **Point sprites**. Use a geometry shader instead. See Shader Stages (Direct3D 10) for additional information.

- **Rasterization rules**. Legacy GDI line rasterization rules are replaced with cleaner, simpler rules. The last-pixel control for lines is no longer supported. See Rasterization Rules (Direct3D 10) for additional information.

- **Shade modes**. D3DSHADEMODE (which support flat/gouraud/phong shading) has been removed. Direct3D 10 implements two interpolations modifiers for vertex shader outputs instead. See FixedFuncEMU sample for an example of emulating Direct3D 9 gouraud and flat shade modes in Direct3D 10.

- **texldp** instruction. An application must implement a projected-texture load with extra HLSL statements. See Reference for HLSL for additional information. FixedFuncEMU sample provides an example of emulating texldp in Direct3D 10.

- **Texture coordinate-index** (TCI) texture-stage state (D3DTSS_TEXCOORDINDEX) is no longer supported.

- **Triangle fans**. An application must convert existing triangle-fans to triangle lists or triangle strips. To emulate some behaviors using DrawPrimitive in older APIs, try using DrawIndexed in Direct3D 10. See Primitive Topologies (Direct3D 10) for additional information.

- **W buffering**. Hardware support is not guaranteed; it is recommended that an application uses high-precision depth buffers instead. See Configuring Depth-Stencil Functionality (Direct3D 10) for additional information.

- **Wrap modes** (texture coordinate wrapping). Texture address wrapping (such as wrap, mirror, clamp etc.) still exist. See **D3D10_SAMPLER_DESC** and **D3D10_TEXTURE_ADDRESS_MODE**.

Related topics

API Features (Direct3D 10)

Pipeline Stages (Direct3D 10)

Article • 01/06/2021 • 2 minutes to read

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

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

- Input-Assembler Stage - The input-assembler stage is responsible for supplying data (triangles, lines and points) to the pipeline.
- Vertex-Shader Stage - The vertex-shader stage processes vertices, typically performing operations such as transformations, skinning, and lighting. A vertex shader always takes a single input vertex and produces a single output vertex.
- Geometry-Shader Stage - The geometry-shader stage processes entire primitives. Its input is a full primitive (which is three vertices for a triangle, two vertices for a line, or a single vertex for a point). In addition, each primitive can also include the vertex data for any edge-adjacent primitives. This could include at

most an additional three vertices for a triangle or an additional two vertices for a line. The Geometry Shader also supports limited geometry amplification and de-amplification. Given an input primitive, the Geometry Shader can discard the primitive, or emit one or more new primitives.

- Stream-Output Stage - The stream-output stage is designed for streaming primitive data from the pipeline to memory on its way to the rasterizer. Data can be streamed out and/or passed into the rasterizer. Data streamed out to memory can be recirculated back into the pipeline as input data or read-back from the CPU.
- Rasterizer Stage - The rasterizer is responsible for clipping primitives, preparing primitives for the pixel shader and determining how to invoke pixel shaders.
- Pixel-Shader Stage - The pixel-shader stage receives interpolated data for a primitive and generates per-pixel data such as color.
- Output-Merger Stage - The output-merger stage is responsible for combining various types of output data (pixel shader values, depth and stencil information) with the contents of the render target and depth/stencil buffers to generate the final pipeline result.

Related topics

Programming Guide for Direct3D 10

Resources (Direct3D 10)

Article • 01/06/2021 • 2 minutes to read

A resource is an area in memory that can be accessed by the Direct3D pipeline. In order for the pipeline to access memory efficiently, data that is provided to the pipeline (such as input geometry, shader resources, textures etc) must be stored in a resource. There are two types of resources from which all Direct3D resources derive: a buffer or a texture. Up to 128 resources can be active for each pipeline stage.

Each application will typically create many resources. Examples of resource include: vertex buffers, index buffer, constant buffer, textures, and shader resources. There are several options that determine how resources can be used. You can create resources that are strongly typed or type less; you can control whether resources have both read and write access; you can make resources accessible to only the CPU, GPU, or both. Naturally, there will be speed vs. functionality tradeoff - the more functionality you allow a resource to have, the less performance you should expect.

Since an application often uses many textures, Direct3D also introduces the concept of a texture array to simplify texture management. A texture array contains one or more textures (all of the same type and dimensions) that can be indexed from within an application or by shaders. Texture arrays allow you to use a single interface with multiple indexes to access many textures. You can create as many texture arrays to manage different texture types as you need.

Once you have created the resources that your application will use, you connect or bind each resource to the pipeline stages that will use them. This is accomplished by calling a bind API, which takes a pointer to the resource. Since more than one pipeline stage may need access to the same resource, Direct3D 10 introduces the concept of a resource view. A view identifies the portion of a resource that can be accessed. You can create m views or a resource and bind them to n pipeline stages, assuming you follow binding rules for shared resource (the runtime will generate errors at compile time if you don't).

A resource view provides a general model for access to a resource (textures, buffers, etc.). Because you can use a view to tell the runtime what data to access and how to access it, resource views allow you create type less resources. That is, you can create resources for a given size at compile time, and then declare the data type within the resource when the resource gets bound to the pipeline. Views expose many new capabilities for using resources, such as the ability to read back depth/stencil surfaces in the shader, generating dynamic cubemaps in a single pass, and rendering simultaneously to multiple slices of a volume.

To find out more about the basic resource types, texture arrays, and how to create and use resources, see these other topics:

- Resource Types
- Choosing a Resource
- Creating Buffer Resources
- Creating Texture Resources
- Copying and Accessing Resource Data
- Memory Structure and Views
- Block Compression
- Table of Resource Limits
- Coordinate Systems
- Floating-Point Rules
- Data Type Conversion Rules
- Mapping Legacy Formats

Related topics

Programming Guide for Direct3D 10

Resource Types (Direct3D 10)

Article • 01/06/2021 • 11 minutes to read

All resources used by the Direct3D pipeline derive from two basic resource types: buffers and textures. A buffer is a collection of raw data (elements); a texture is a collection of texels (texture elements).

- Buffer Resources
  - Buffer Types
- Texture Resources
  - Texture Types
  - Subresources
  - Strong vs. Weak Typing
- Related topics

There are two ways to fully specify the layout (or memory footprint) of a resource:

Item	Description
Typed	Fully specify the type when the resource is created.
Typeless	Fully specify the type when the resource is bound to the pipeline.

Buffer Resources

A buffer resource is a collection of fully typed data; internally, a buffer contains elements. An element is made up of 1 to 4 components. Examples of element data types include: a packed data value (like R8G8B8A8), a single 8-bit integer, four 32-bit float values. These data types are used to store data, such as a position vector, a normal vector, a texture coordinate in a vertex buffer, an index in an index buffer, or device state.

A buffer is created as an unstructured resource. Because it is unstructured, a buffer cannot contain any mipmap levels, is not filtered when read, and cannot be multisampled.

Buffer Types

- Vertex Buffer

- Index Buffer
- Constant Buffer

Vertex Buffer

A buffer is a collection of elements; a vertex buffer contains per-vertex data. The simplest example is a vertex buffer that contains one type of data, such as 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 for eight vertices; 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 and these other buffer parameters:

- Offset - the number of bytes from the start of the buffer to the data for the first vertex. The offset is supplied to **IASetVertexBuffers**.
- BaseVertexLocation - the number of bytes from the offset to the first vertex used by the appropriate draw call (see Draw Methods).

Before you create a vertex buffer, you need to define its layout by creating an **input-layout object**; this is done by calling **CreateInputLayout**. Once the input-layout object is created, bind it to the input-assembler stage by calling **IASetInputLayout**.

To create a vertex buffer, call **CreateBuffer**.

Index Buffer

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. Using an index buffer with one or more vertex buffers to supply data to the IA stage is called indexing. An index buffer can be visualized like the following illustration.

The sequential indices stored in an index buffer are located with the following parameters:

- Offset - the number of bytes from the start of the buffer to the first index. The offset is supplied to **IASetIndexBuffer**.
- StartIndexLocation - the number of bytes from the offset to the first vertex used by the appropriate draw call (see Draw Methods).
- IndexCount - the number of indices to render.

To create an index buffer, call **CreateBuffer**.

An index buffer can stitch together multiple line or triangle strips by separating each with a strip-cut index. A strip-cut index allows multiple line or triangle strips to be drawn with a single draw call. A strip-cut index is simply the maximum possible value for the index (0xffff for a 16-bit index, 0xffffffff for a 32-bit index). The strip-cut index resets the winding order in indexed primitives and can be used to remove the need for degenerate triangles that may otherwise be required to maintain proper winding order in a triangle strip. The following illustration shows an example of a strip-cut index.

Constant Buffer

Direct3D 10 introduced a new buffer for supplying shader constants called a shader-constant buffer or simply a constant buffer. Conceptually, it 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.

Constant buffers reduce the bandwidth required to update shader constants by allowing shader constants to be grouped together and committed at the same time rather than making individual calls to commit each constant separately.

To create a shader-constant buffer, call **CreateBuffer** and specify the constant-buffer bind flag D3D10_BIND_CONSTANT_BUFFER (see **D3D10_BIND_FLAG**).

To bind a shader-constant buffer to the pipeline, call one of these methods: **GSSetConstantBuffers**, **PSSetConstantBuffers**, or **VSSetConstantBuffers**.

Note that when using the **ID3D10Effect Interface** interface the process of creating, binding and comitting a constant buffer is handled by the **ID3D10Effect Interface** instance. In that case it is only nescesary to get the variable from the effect with one of the GetVariable methods such as **GetVariableByName** and update the variable with one of the SetVariable methods such as **SetMatrix**. For an example of using **ID3D10Effect Interface** to manage a constant buffer see tutorial 07.

A shader continues to read variables in a constant buffer directly by variable name in the same manner variables that are not part of a constant buffer are read.

Each shader stage allows up to 15 shader-constant buffers; each buffer can hold up to 4096 constants.

Use a constant buffer to store the results of the stream-output stage.

See Shader Constants (DirectX HLSL) for an example of declaring a constant buffer in a shader.

Texture Resources

A texture resource is a structured collection of data designed to store texels. Unlike buffers, textures can be filtered by texture samplers as they are read by shader units. The type of texture impacts how the texture is filtered. A texel represents the smallest unit of a texture that can be read or written to by the pipeline. Each texel contains 1 to 4 components, arranged in one of the DXGI formats (see **DXGI_FORMAT**).

Textures are created as a structured resource so that their size is known. However, each texture may be typed or type less at resource-creation time, as long as the type is fully specified using a view when the texture is bound to the pipeline.

- Texture Types
- Subresources
- Strong vs. Weak Typing

Texture Types

There are several types of textures: 1D, 2D, 3D, each of which can be created with or without mipmaps. Direct3D 10 also supports texture arrays and multisampled textures.

- 1D Texture
- 1D Texture Array
- 2D Texture and 2D Texture Array
- 3D Texture

1D Texture

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.

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 top-most level contains the most detail, each subsequent level is smaller; for a 1D mipmap, the smallest level contains one texel. The differing levels are identified by an index called a LOD (level-of-detail); you can use the LOD to access a smaller texture when rendering geometry that is not as close to the camera.

1D Texture Array

Direct3D 10 also has a new data structure for an array of textures. 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 Texture and 2D Texture Array

A Texture2D resource contains a 2D grid of texels. Each texel is addressable by a u, v vector. Because it is a texture resource, it may contain mipmap levels, and subresources. A fully populated 2D texture resource looks like the following illustration.

This texture resource contains a single 3x5 texture with three mipmap levels.

A Texture2DArray resource is a homogeneous array of 2D textures; that is, each texture has the same data format and dimensions (including mipmap levels). It has a similar layout as the 1D texture array except that the textures now contain 2D data, and therefore looks like the following illustration.

This texture array contains three textures; each texture is 3x5 with two mipmap levels.

Using a Texture2DArray as a Texture Cube

A texture cube is a 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.

3D Texture

A Texture3D resource (also known as a volume texture) contains a 3D volume of texels. Since it is a texture resource, it may contain mipmap levels. A fully populated **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.

Subresources

The Direct3D 10 API references entire resources or subsets of resources. To specify portion of resources, Direct3D has coined the term subresources which means a subset of a resource.

A buffer is defined as a single subresource. Textures are a little more complicated as 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 this.

Think of this as a single texture that is made up of three subtextures. Each subtexture is counted as a subresource, so this 1D texture contains 3 subresources. A subtexture (or subresource) can be indexed using the level-of-detail (LOD) for a single texture. When using an array of textures, accessing a particular subtexture 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 here.

Selecting Subresources

Some API's access an entire resource (for example **CopyResource**), others access a portion of a resource (for example **UpdateSubresource** or **CopySubresourceRegion**). The API's that access a portion of a resource generally use a view description (such as **D3D10_TEX2D_ARRAY_DSV**) to specify the subresources to access.

These figures illustrate 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 subtextures, 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.

Regardless of what texture type you are using, with or without mipmaps, with or without a texture array, you can use the helper function, **D3D10CalcSubresource**, to compute the index of a particular subresource.

Strong vs. Weak Typing

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.

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 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 component bit of the new format matches the bit count of the old format.

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 an DXGI_FORMAT_R32G32B32A32_FLOAT and an DXGI_FORMAT_R32G32B32A32_UINT at different locations in the pipeline simultaneously.

Related topics

Resources (Direct3D 10)

Choosing a Resource (Direct3D 10)

Article • 01/06/2021 • 4 minutes to read

A resource is a collection of data that is used by the 3D pipeline. Creating resources and defining their behavior is the first step toward programming your application. This guide covers basic topics for choosing the resources required by your application.

- Identify Pipeline Stages That Need Resources
- Identify How Each Resource Will Be Used
- Binding Resources to Pipeline Stages
- Related topics

Identify Pipeline Stages That Need Resources

The first step is to choose the pipeline stage (or stages) that will use a resource. That is, identify each stage that will read data from a resource as well as the stages that will write data out to a resource. Knowing the pipeline stages where the resources will be used determines the APIs that will be called to bind the resource to the stage.

This table lists the types of resources that can be bound to each pipeline stage. It includes whether the resource can be bound as an input or an output, as well as the bind API.

Pipeline Stage	In/Out	Resource	Resource Type	Bind API
Input Assembler	In	Vertex Buffer	Buffer	IASetVertexBuffers
Input Assembler	In	Index Buffer	Buffer	IASetIndexBuffer
Shader Stages	In	Shader-ResourceView	Buffer, Texture1D, Texture2D, Texture3D	VSSetShaderResources, GSSetShaderResources, PSSetShaderResources
Shader Stages	In	Shader-Constant Buffer	Buffer	VSSetConstantBuffers, GSSetConstantBuffers, PSSetConstantBuffers
Stream Output	Out	Buffer	Buffer	SOSetTargets

Pipeline Stage	In/Out	Resource	Resource Type	Bind API
Output Merger	Out	Render-target View	Buffer, Texture1D, Texture2D, Texture3D	OMSetRenderTargets
Output Merger	Out	Depth/Stencil View	Texture1D, Texture2D	OMSetRenderTargets

Identify How Each Resource Will Be Used

Once you have chosen the pipeline stages that your application will use (and therefore the resources that each stage will require), the next step is to determine how each resource will be used, that is, whether a resource can be accessed by the CPU or the GPU.

The hardware that your application is running on will have a minimum of at least one CPU and one GPU. To pick a usage value, consider which type of processor needs to read or write to the resource from the following options (see D3D10_USAGE).

Resource Usage	Can be updated by	Frequency of Update
Default	GPU	infrequently
Dynamic	CPU	frequently
Staging	GPU	n/a
Immutable	CPU (only at resource creation time)	n/a

Default usage should be used for a resource that is expected to be updated by the CPU infrequently (less than once per frame). Ideally, the CPU would never write directly to a resource with default usage so as to avoid potential performance penalties.

Dynamic usage should be used for a resource that the CPU updates relatively frequently (once or more per frame). A typical scenario for a dynamic resource would be to create dynamic vertex and index buffers that would be filled at runtime with data about the geometry visible from the point of view of the user for each frame. These buffers would be used to render only the geometry visible to the user for that frame.

Staging usage should be used to copy data to and from other resources. A typical scenario would be to copy data in a resource with default usage (which the CPU cannot access) to a resource with staging usage (which the CPU can access).

Immutable resources should be used when the data in the resource will never change.

Another way of looking at the same idea is to think of what an application does with a resource.

How Application uses the Resource	Resource Usage
Load once and never update	Immutable or Default
Application fills resource repeatedly	Dynamic
Render to texture	Default
CPU access of GPU data	Staging

If you are unsure what usage to choose, start with the default usage as it is expected to be the most common case. A Shader-Constant buffer is the one resource type that should always have default usage.

Binding Resources to Pipeline Stages

A resource can be bound to more than one pipeline stage at the same time as long as the restrictions specified when the resource was created (**usage flags**, **bind flags**, **cpu access flags**) are met. More specifically, a resource can be bound as an input and an output simultaneously as long as reading and writing part of a resource cannot happen at the same time.

When binding a resource, think about how the GPU and the CPU will access the resource. Resources that are designed for a single purpose (do not use multiple usage, bind, and cpu access flags) will more than likely result in better performance.

For example, consider the case of a render target used as a texture multiple times. It may be faster to have two resources: a render target and a texture used as a shader resource. Each resource would use only one bind flag (**D3D10_BIND_RENDER_TARGET** or **D3D10_BIND_SHADER_RESOURCE**). The data would be copied from the render-target texture to the shader texture using **CopyResource** or **CopySubresourceRegion**. This may improve performance by isolating the render-target write from the shader-

texture read. Of course, the only way to be sure is to implement both approaches and measure the performance difference in your particular application.

Related topics

Resources (Direct3D 10)

Creating Buffer Resources (Direct3D 10)

Article • 01/06/2021 • 4 minutes to read

Creating buffers requires defining the data that the buffer will store, providing initialization data, and setting up appropriate usage and binding flags. To create textures, see Creating Texture Resources (Direct3D 10).

- Create a Vertex Buffer
  - Create a Buffer Description
  - Create the Initialization Data for the Buffer
  - Create the Buffer
- Create an Index Buffer
- Create a Constant Buffer
- Related topics

Create a Vertex Buffer

The steps to creating a vertex buffer are as follows.

- Create a Buffer Description
- Create the Initialization Data for the Buffer
- Create the Buffer

Create a Buffer Description

When creating a vertex buffer, a buffer description (see **D3D10_BUFFER_DESC**) is used to define how data is organized within the buffer, how the pipeline can access the buffer, and how the buffer will be used.

The following example demonstrates how to create a buffer description for a single triangle with vertices that contain position and color values.

```
struct SimpleVertex
{
    D3DXVECTOR3 Position;
    D3DXVECTOR3 Color;
};

    D3D10_BUFFER_DESC bufferDesc;
    bufferDesc.Usage            = D3D10_USAGE_DEFAULT;
    bufferDesc.ByteWidth        = sizeof( SimpleVertex ) * 3;
```

```
    bufferDesc.BindFlags        = D3D10_BIND_VERTEX_BUFFER;
    bufferDesc.CPUAccessFlags   = 0;
    bufferDesc.MiscFlags        = 0;
```

In this example, the buffer description is initialized with almost all default settings for usage, CPU access and miscellaneous flags. The other settings are for the bind flag which identifies the resource as a vertex buffer only, and the size of the buffer.

The usage and CPU access flags are important for performance. These two flags together determine how often a resource gets accessed, what type of memory the resource could be loaded into, and what processor needs to access the resource. Default usage this resource will not be updated very often. Setting CPU access to 0 means that the CPU will not need to either read or write the resource. Taken in combination, this means that the runtime can load the resource into the highest performing memory for the GPU since the resource does not require CPU access.

As expected, there is a tradeoff between best performance and any-time accessibility by either processor. For example, the default usage with no CPU access means that the resource can be made available to the GPU exclusively. This could include loading the resource into memory not directly accessible by the CPU. The resource could only be modified with UpdateSubresource.

Create the Initialization Data for the Buffer

A buffer is just a collection of elements and is laid out as a 1D array. As a result, the system memory pitch and system memory slice pitch are both the same; the size of the vertex data declaration. An application can provide initialization data when a buffer is created using a subresource description, which contains a pointer to the actual resource data and contains information about the size and layout of that data.

Any buffer created with immutable usage (see D3D10_USAGE_IMMUTABLE) must be initialized at creation time. Buffers that use any of the other usage flags can be updated after initialization using CopyResource, CopySubresourceRegion, and UpdateSubresource, or by accessing its underlying memory using the Map method.

Create the Buffer

Using the buffer description and the initialization data (which is optional) call CreateBuffer to create a vertex buffer. The following code snippet demonstrates how to create a vertex buffer from an array of vertex data declared by the application.

```
struct SimpleVertexCombined
{
    D3DXVECTOR3 Pos;
    D3DXVECTOR3 Col;
};


ID3D10InputLayout* g_pVertexLayout = NULL;
ID3D10Buffer*      g_pVertexBuffer[2] = { NULL, NULL };
ID3D10Buffer*      g_pIndexBuffer = NULL;


SimpleVertexCombined verticesCombo[] =
{
    D3DXVECTOR3( 0.0f, 0.5f, 0.5f ),
    D3DXVECTOR3( 0.0f, 0.0f, 0.5f ),
    D3DXVECTOR3( 0.5f, -0.5f, 0.5f ),
    D3DXVECTOR3( 0.5f, 0.0f, 0.0f ),
    D3DXVECTOR3( -0.5f, -0.5f, 0.5f ),
    D3DXVECTOR3( 0.0f, 0.5f, 0.0f ),
};


    D3D10_BUFFER_DESC bufferDesc;
    bufferDesc.Usage            = D3D10_USAGE_DEFAULT;
    bufferDesc.ByteWidth        = sizeof( SimpleVertexCombined ) * 3;
    bufferDesc.BindFlags        = D3D10_BIND_VERTEX_BUFFER;
    bufferDesc.CPUAccessFlags   = 0;
    bufferDesc.MiscFlags        = 0;

    D3D10_SUBRESOURCE_DATA InitData;
    InitData.pSysMem = verticesCombo;
    InitData.SysMemPitch = 0;
    InitData.SysMemSlicePitch = 0;
    hr = g_pd3dDevice->CreateBuffer( &bufferDesc, &InitData,
&g_pVertexBuffer[0] );
```

Create an Index Buffer

Creating an index buffer is very similar to creating a vertex buffer; with two differences. An index buffer contains only 16-bit or 32-bit data (instead of the wide range of formats available to a vertex buffer. An index buffer also requires an index-buffer bind flag.

The following example demonstrates how to create an index buffer from an array of index data.

```

ID3D10Buffer *g_pIndexBuffer = NULL;
```

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

    D3D10_BUFFER_DESC bufferDesc;
    bufferDesc.Usage           = D3D10_USAGE_DEFAULT;
    bufferDesc.ByteWidth       = sizeof( unsigned int ) * 3;
    bufferDesc.BindFlags       = D3D10_BIND_INDEX_BUFFER;
    bufferDesc.CPUAccessFlags  = 0;
    bufferDesc.MiscFlags       = 0;

    D3D10_SUBRESOURCE_DATA InitData;
    InitData.pSysMem = indices;
    InitData.SysMemPitch = 0;
    InitData.SysMemSlicePitch = 0;
    hr = g_pd3dDevice->CreateBuffer( &bufferDesc, &InitData, &g_pIndexBuffer
);
    if( FAILED( hr ) )
        return hr;

    g_pd3dDevice->IASetIndexBuffer( g_pIndexBuffer, DXGI_FORMAT_R32_UINT, 0
);
```

Create a Constant Buffer

Direct3D 10 introduces a constant buffer. A constant buffer, or shader constant buffer, is a buffer that contains shader constants. Here is an example of creating a constant buffer, taken from the HLSLWithoutFX10 Sample.

```

ID3D10Buffer*   g_pConstantBuffer10 = NULL;

struct VS_CONSTANT_BUFFER
{
    D3DXMATRIX mWorldViewProj;      //mWorldViewProj will probably be global
to all shaders in a project.
                                    //It's a good idea not to move it around
between shaders.
    D3DXVECTOR4 vSomeVectorThatMayBeNeededByASpecificShader;
    float fSomeFloatThatMayBeNeededByASpecificShader;
    float fTime;                    //fTime may also be global to all
shaders in a project.
    float fSomeFloatThatMayBeNeededByASpecificShader2;
    float fSomeFloatThatMayBeNeededByASpecificShader3;
};

    D3D10_BUFFER_DESC cbDesc;
    cbDesc.ByteWidth = sizeof( VS_CONSTANT_BUFFER );
    cbDesc.Usage = D3D10_USAGE_DYNAMIC;
    cbDesc.BindFlags = D3D10_BIND_CONSTANT_BUFFER;
    cbDesc.CPUAccessFlags = D3D10_CPU_ACCESS_WRITE;
```

```
    cbDesc.MiscFlags = 0;
    hr = g_pd3dDevice->CreateBuffer( &cbDesc, NULL, &g_pConstantBuffer10 );
    if( FAILED( hr ) )
       return hr;

    g_pd3dDevice->VSSetConstantBuffers( 0, 1, g_pConstantBuffer10 );
```

Note that when using the **ID3D10Effect Interface** interface the process of creating, binding and comitting a constant buffer is handled by the **ID3D10Effect Interface** instance. In that case it is only nescesary to get the variable from the effect with one of the GetVariable methods such as **GetVariableByName** and update the variable with one of the SetVariable methods such as **SetMatrix**. For an example of using **ID3D10Effect Interface** to manage a constant buffer see Tutorial 7: Texture Mapping and Constant Buffers.

Related topics

Resources (Direct3D 10)

Creating Texture Resources (Direct3D 10)

Article • 01/06/2021 • 9 minutes to read

A texture resource is a structured collection of data. Typically, color values are stored in textures and accessed during rendering by the pipeline at various stages for both input and output. Creating textures and defining how they will be used is an important part of rendering interesting-looking scenes in Direct3D 10.

Even though textures typically contain color information, creating textures using different **DXGI_FORMAT** enables them to store different kinds of data. This data can then be leveraged by the Direct3D 10 pipeline in non-traditional ways.

All textures have limits on how much memory they consume, and how many texels they contain. These limits are specified by resource constants.

- Create a Texture from a File
  - Create Texture and View Separately
  - Create Texture and View Simultaneously
- Creating Empty Textures
  - Rendering to a texture
  - Filling Textures Manually
- Multiple Rendertargets
- Related topics

Create a Texture from a File

> **Note**
>
> The **D3DX utility library** is deprecated for Windows 8 and is not supported for Windows Store apps.

When you create a texture in Direct3D 10, you also need to create a view. A view is an object that tells the device how a texture should be accessed during rendering. The most common way to access a texture is to read from it using a shader. A shader-resource view will tell a shader how to read from a texture during rendering. The kind of view a texture will use must be specified when you create it.

Creating a texture and loading its initial data can be done in two different ways: create a texture and view separately, or create both the texture and the view at the same time. The API provides both techniques so you can choose which better suits your needs.

Create Texture and View Separately

The easiest way to create a texture is to load it from an image file. To create the texture, just fill in one structure and provide the texture's name to D3DX10CreateTextureFromFile.

```
ID3D10Device *pDevice = NULL;
// Initialize D3D10 device...

D3DX10_IMAGE_LOAD_INFO loadInfo;
ZeroMemory( &loadInfo, sizeof(D3DX10_IMAGE_LOAD_INFO) );
loadInfo.BindFlags = D3D10_BIND_SHADER_RESOURCE;

ID3D10Resource *pTexture = NULL;
D3DX10CreateTextureFromFile( pDevice, L"sample.bmp", &loadInfo, NULL,
&pTexture, NULL );
```

The D3DX function D3DX10CreateTextureFromFile does three things: first, it creates a Direct3D 10 texture object; second, it reads the input image file; third, it stores the image data in the texture object. In the sample above, a BMP file is loaded, but the function can load a variety of file types.

The bind flag indicates the texture will be created as a shader resource which allows a shader stage to read from the texture during rendering.

The above example does not specify all of the loading parameters. In fact, it's often beneficial to simply zero out the loading parameters as this enables D3DX to choose appropriate values based on the input image. If you want the input image to determine all of the parameters the texture is created with, simply specify **NULL** for the **loadInfo** parameter like this:

```
D3DX10CreateTextureFromFile( pDevice, L"sample.bmp", NULL, NULL, &pTexture,
NULL );
```

Specifying **NULL** for the loading information is a simple yet powerful shortcut.

Now that a texture has been created, you need to create a shader-resource view so that the texture can be bound as an input to a shader. Since D3DX10CreateTextureFromFile returns a pointer to a resource and not a pointer to a texture, you have to determine the exact type of resource that was loaded, and then you can create a shader-resource view using CreateShaderResourceView.

```
D3D10_SHADER_RESOURCE_VIEW_DESC srvDesc;
D3D10_RESOURCE_DIMENSION type;
pTexture->GetType( &type );
switch( type )
{
    case D3D10_RESOURCE_DIMENSION_BUFFER:
    //...
    break;
    case D3D10_RESOURCE_DIMENSION_TEXTURE1D:
    //...
    break;
    case D3D10_RESOURCE_DIMENSION_TEXTURE2D:
    {
        D3D10_TEXTURE2D_DESC desc;
        ID3D10Texture2D *pTexture2D = (ID3D10Texture2D*)pTexture;
        pTexture2D->GetDesc( &desc );

        srvDesc.Format = desc.Format;
        srvDesc.ViewDimension = D3D10_SRV_DIMENSION_TEXTURE2D;
        srvDesc.Texture2D.MipLevels = desc.MipLevels;
        srvDesc.Texture2D.MostDetailedMip = desc.MipLevels -1;

    }
    break;
    case D3D10_RESOURCE_DIMENSION_TEXTURE3D:
    //...
    break;
    default:
    //...
    break;
}

ID3D10ShaderResourceView *pSRView = NULL;
pDevice->CreateShaderResourceView( pTexture, &srvDesc, &pSRView );
```

Although the above sample creates a 2D shader-resource view, the code to create other shader-resource view types is very similar. Any shader stage can now use this texture as an input.

Using D3DX10CreateTextureFromFile and CreateShaderResourceView to create a texture and its associated view is one way to prepare a texture to be bound to a shader

stage. Another way to do so is to create both the texture and its view at the same time, which is discussed in the next section.

Create Texture and View Simultaneously

Direct3D 10 requires both a texture and a shader-resource view to read from a texture during runtime. Since the creation of a texture and a shader-resource view is such a common task, D3DX provides the **D3DX10CreateShaderResourceViewFromFile** to do it for you.

```
D3DX10_IMAGE_LOAD_INFO loadInfo;
ZeroMemory( &loadInfo, sizeof(D3DX10_IMAGE_LOAD_INFO) );
loadInfo.BindFlags = D3D10_BIND_SHADER_RESOURCE;
loadInfo.Format = DXGI_FORMAT_BC1_UNORM;

ID3D10ShaderResourceView *pSRView = NULL;
D3DX10CreateShaderResourceViewFromFile( pDevice, L"sample.bmp", &loadInfo,
NULL, &pSRView, NULL );
```

With a single D3DX call, both the texture and shader-resource view are created. The functionality of the **loadInfo** parameter is unchanged; you can use it to customize how the texture is created, or derive the necessary parameters from the input file by specifying **NULL** for the **loadInfo** parameter.

The **ID3D10ShaderResourceView** object returned by the **D3DX10CreateShaderResourceViewFromFile** function can later be used to retrieve the original **ID3D10Resource** interface if it is needed. This can be done by calling the **GetResource** method.

D3DX provides a single function to create a texture and shader-resource view as a convienience; it is up to you to decide which method of creating a texture and view best suits the needs of your application.

Now that you know how to create a texture and its shader-resource view, the next section will show you how you can sample (read) from that texture using a shader.

Creating Empty Textures

Sometimes applications will want to create a texture and compute the data to be stored in the texture, or use the graphics pipeline to render to this texture and later use the results in other processing. These textures could be updated by the graphics pipeline or

by the application itself, depending on what kind of **usage** was specified for the texture when it was created.

Rendering to a texture

The most common case of creating an empty texture to be filled with data during runtime is the case where an application wants to render to a texture and then use the results of the rendering operation in a subsequent pass. Textures created with this purpose should specify default **usage**.

The following code sample creates an empty texture that the pipeline can render to and subsequently use as an input to a shader.

```
// Create the render target texture
D3D10_TEXTURE2D_DESC desc;
ZeroMemory( &desc, sizeof(desc) );
desc.Width = 256;
desc.Height = 256;
desc.MipLevels = 1;
desc.ArraySize = 1;
desc.Format = DXGI_FORMAT_R32G32B32A32_FLOAT;
desc.SampleDesc.Count = 1;
desc.Usage = D3D10_USAGE_DEFAULT;
desc.BindFlags = D3D10_BIND_RENDER_TARGET | D3D10_BIND_SHADER_RESOURCE;

ID3D10Texture2D *pRenderTarget = NULL;
pDevice->CreateTexture2D( &desc, NULL, &pRenderTarget );
```

Creating the texture requires the application to specify some information about the properties the texture will have. The width and height of the texture in texels is set to 256. For this render target, a single mipmap level is all we need. Only one render target is required so the array size is set to 1. Each texel contains four 32-bit floating point values, which can be used to store very precise information (see **DXGI_FORMAT**). One sample per pixel is all that will be needed. The usage is set to default because this allows for the most efficient placement of the render target in memory. Finally, the fact that the texture will be bound as a render target and a shader resource at different points in time is specified.

Textures cannot be bound for rendering to the pipeline directly; use a render-target view as shown in the following code sample.

```
D3D10_RENDER_TARGET_VIEW_DESC rtDesc;
rtDesc.Format = desc.Format;
rtDesc.ViewDimension = D3D10_RTV_DIMENSION_TEXTURE2D;
rtDesc.Texture2D.MipSlice = 0;

ID3D10RenderTargetView *pRenderTargetView = NULL;
pDevice->CreateRenderTargetView( pRenderTarget, &rtDesc, &pRenderTargetView
);
```

The format of the render-target view is simply set to the format of the original texture. The information in the resource should be interpreted as a 2D texture, and we only want to use the first mipmap level of the render target.

Similarly to how a render-target view must be created so that the render target can be bound for output to the pipeline, a shader-resource view must be created so that the render target can be bound to the pipeline as an input. The following code sample demonstrates this.

```
// Create the shader-resource view
D3D10_SHADER_RESOURCE_VIEW_DESC srDesc;
srDesc.Format = desc.Format;
srDesc.ViewDimension = D3D10_SRV_DIMENSION_TEXTURE2D;
srDesc.Texture2D.MostDetailedMip = 0;
srDesc.Texture2D.MipLevels = 1;

ID3D10ShaderResourceView *pShaderResView = NULL;
pDevice->CreateShaderResourceView( pRenderTarget, &srDesc, &pShaderResView
);
```

The parameters of shader-resource view descriptions are very similar to those of render-target view descriptions and were chosen for the same reasons.

Filling Textures Manually

Sometimes applications would like to compute values at runtime, put them into a texture manually and then have the graphics pipeline use this texture in later rendering operations. To do this, the application must create an empty texture in such a way to allow the CPU to access the underlying memory. This is done by creating a dynamic texture and gaining access to the underlying memory by calling a particular method. The following code sample demonstrates how to do this.

```
D3D10_TEXTURE2D_DESC desc;
desc.Width = 256;
desc.Height = 256;
desc.MipLevels = desc.ArraySize = 1;
desc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
desc.SampleDesc.Count = 1;
desc.Usage = D3D10_USAGE_DYNAMIC;
desc.BindFlags = D3D10_BIND_SHADER_RESOURCE;
desc.CPUAccessFlags = D3D10_CPU_ACCESS_WRITE;
ID3D10Texture2D *pTexture = NULL;
pd3dDevice->CreateTexture2D( &desc, NULL, &pTexture );
```

Note that the format is set to a 32 bits per pixel where each component is defined by 8 bits. The usage parameter is set to dynamic while the bind flags are set to specify the texture will be accessed by a shader. The rest of the texture description is similar to creating a render target.

Calling Map enables the application to access the underlying memory of the texture. The pointer retrieved is then used to fill the texture with data. This can be seen in the following code sample.

```
D3D10_MAPPED_TEXTURE2D mappedTex;
pTexture->Map( D3D10CalcSubresource(0, 0, 1), D3D10_MAP_WRITE_DISCARD, 0,
&mappedTex );

UCHAR* pTexels = (UCHAR*)mappedTex.pData;
for( UINT row = 0; row < desc.Height; row++ )
{
    UINT rowStart = row * mappedTex.RowPitch;
    for( UINT col = 0; col < desc.Width; col++ )
    {
        UINT colStart = col * 4;
        pTexels[rowStart + colStart + 0] = 255; // Red
        pTexels[rowStart + colStart + 1] = 128; // Green
        pTexels[rowStart + colStart + 2] = 64;  // Blue
        pTexels[rowStart + colStart + 3] = 32;  // Alpha
    }
}

pTexture->Unmap( D3D10CalcSubresource(0, 0, 1) );
```

Multiple Rendertargets

Up to eight render-target views can be bound to the pipeline at a time (with OMSetRenderTargets). For each pixel (or each sample if multisampling is enabled),

blending is done independently for each render-target view. Two of the blend state variables - BlendEnable and RenderTargetWriteMask - are arrays of eight, each array member corresponds to a render-target view. When using multiple render targets, each render target must be the same resource type (buffer, 1D texture, 2D texture array, etc.) and must have the same dimension (width, height, depth for 3D textures, and array size for texture arrays). If the render targets are multisampled, then they must all have the same number of samples per pixel.

There can only be one depth-stencil buffer active, regardless of how many render targets are active. When using texture arrays as render targets, all view dimensions must match. The render targets do not need to have the same texture format.

Related topics

Resources (Direct3D 10)

Copying and Accessing Resource Data (Direct3D 10)

Article • 08/19/2021 • 8 minutes to read

It is no longer necessary to think about resources as being created in either video memory or system memory. Or whether or not the runtime should manage the memory. Thanks to the architecture of the new WDDM (Windows Display Driver Model), applications now create Direct3D 10 resources with different **usage** flags to indicate how the application intends on using the resource data. The new driver model virtualizes the memory used by resources; it then becomes the responsibility of the operating system/driver/memory manager to place resources in the most performant area of memory possible given the expected usage.

The default case is for resources to be available to the GPU. Of course, having said that, there are times when the resource data needs to be available to the CPU. Copying resource data around so that the appropriate processor can access it without impacting performance requires some knowledge of how the API methods work.

- Copying Resource Data
- Accessing Resource Data

Copying Resource Data

Resources are created in memory when Direct3D executes a Create call. They can be created in video memory, system memory, or any other kind of memory. Since WDDM driver model virtualizes this memory, applications no longer need to keep track of what kind of memory resources are created in.

Ideally, all resources would be located in video memory so that the GPU can have immediate access to them. However, it is sometimes necessary for the CPU to read the resource data or for the GPU to access resource data the CPU has written to. Direct3D 10 handles these different scenarios by requesting the application specify a usage, and then offers several methods for copying resource data when necessary.

Depending on how the resource was created, it is not always possible to directly access the underlying data. This may mean that the resource data must be copied from the source resource to another resource that is accessible by the appropriate processor. In terms of Direct3D 10, default resources can be accessed directly by the GPU, dynamic and staging resources can be directly accessed by the CPU.

Once a resource has been created, its **usage** cannot be changed. Instead, copy the contents of one resource to another resource that was created with a different usage. Direct3D 10 provides this functionality with three different methods. The first two methods(**ID3D10Device::CopyResource** and **ID3D10Device::CopySubresourceRegion**) are designed to copy resource data from one resource to another. The third method (**ID3D10Device::UpdateSubresource**) is designed to copy data from memory to a resource.

There are two main kinds of resources: mappable and non-mappable. Resources created with dynamic or staging usages are mappable, while resources created with default or immutable usages are non-mappable.

Copying data among non-mappable resources is very fast because this is the most common case and has been optimized to perform well. Since these resources are not directly accessible by the CPU, they are optimized so that the GPU can manipulate them quickly.

Copying data among mappable resources is more problematic because the performance will depend on the usage the resource was created with. For example, the GPU can read a dynamic resource fairly quickly but cannot write to them, and the GPU cannot read or write to staging resources directly.

Applications that wish to copy data from a resource with default usage to a resource with staging usage (to allow the CPU to read the data -- i.e. the GPU readback problem) must do so with care. See Accessing Resource Data for more details on this last case.

Accessing Resource Data

Accessing a resource requires mapping the resource; mapping essentially means the application is trying to give the CPU access to memory. The process of mapping a resource so that the CPU can access the underlying memory can cause some performance bottlenecks and for this reason, care must be taken as to how and when to perform this task.

Performance can grind to a halt if the application tries to map a resource at the wrong time. If the application tries to access the results of an operation before that operation is finished, a pipeline stall will occur.

Performing a map operation at the wrong time could potentially cause a severe drop in performance by forcing the GPU and the CPU to synchronize with each other. This synchronization will occur if the application wants to access a resource before the GPU is finished copying it into a resource the CPU can map.

The CPU can only read from resources created with the D3D10_USAGE_STAGING flag. Since resources created with this flag cannot be set as outputs of the pipeline, if the CPU wants to read the data in a resource generated by the GPU, the data must be copied to a resource created with the staging flag. The application may do this by using the **ID3D10Device::CopyResource** or **ID3D10Device::CopySubresourceRegion** methods to copy the contents of one resource to another. The application can then gain access to this resource by calling the appropriate Map method. When access to the resource is no longer needed, the application should then call the corresponding Unmap method. For example, **ID3D10Texture2D::Map** and **ID3D10Texture2D::Unmap**. The different Map methods return some specific values depending on the input flags. See **Map Remarks section** for details.

> ⓘ **Note**
>
> When the application calls the Map method, it receives a pointer to the resource data to access. The runtime ensures that the pointer has a specific alignment, depending on **feature level**. For **D3D_FEATURE_LEVEL_10_0** and higher, the pointer is aligned to 16 bytes. For lower than **D3D_FEATURE_LEVEL_10_0**, the pointer is aligned to 4 bytes. The 16-byte alignment allows the application to perform **SSE**-optimized operations on the data natively, without realignment or copy.

Performance Considerations

It is best to think of a PC as a machine running as a parallel architecture with two main types of processors: one or more CPU's and one or more GPU's. As in any parallel architecture, the best performance is achieved when each processor is scheduled with enough tasks to prevent it from going idle and when the work of one processor is not waiting on the work of another.

The worst-case scenario for GPU/CPU parallelism is the need to force one processor to wait for the results of work done by another. Direct3D 10 tries to remove this cost by making the **ID3D10Device::CopyResource** and **ID3D10Device::CopySubresourceRegion** methods asynchronous; the copy has not necessarily executed by the time the method returns. The benefit of this is that the application does not pay the performance cost of actually copying the data until the CPU accesses the data, which is when Map is called. If the Map method is called after the data has actually been copied, no performance loss occurs. On the other hand, if the Map method is called before the data has been copied, then a pipeline stall will occur.

Asynchronous calls in Direct3D 10 (which are the vast majority of methods, and especially rendering calls) are stored in what is called a command buffer. This buffer is internal to the graphics driver and is used to batch calls to the underlying hardware so that the costly switch from user mode to kernel mode in Microsoft Windows occurs as rarely as possible.

The command buffer is flushed, thus causing a user/kernel mode switch, in one of four situations, which are as follows.

1. **Present** is called.
2. **ID3D10Device::Flush** is called.
3. The command buffer is full; its size is dynamic and is controlled by the Operating System and the graphics driver.
4. The CPU requires access to the results of a command waiting to execute in the command buffer.

Of the four situations above, number four is the most critical to performance. If the application issues a **ID3D10Device::CopyResource** or **ID3D10Device::CopySubresourceRegion** call, this call is queued in the command buffer. If the application then tries to map the staging resource that was the target of the copy call before the command buffer has been flushed, a pipeline stall will occur because not only does the Copy method call need to execute, but all other buffered commands in the command buffer must execute as well. This will cause the GPU and CPU to synchronize because the CPU will be waiting to access the staging resource while the GPU is emptying the command buffer and finally filling the resource the CPU needs. Once the GPU finishes the copy, the CPU will begin accessing the staging resource, but during this time, the GPU will be sitting idle.

Doing this frequently at runtime will severely degrade performance. For that reason, mapping of resources created with default usage should be done with care. The application needs to wait long enough for the command buffer to be emptied and thus have all of those commands finish executing before it tries to map the corresponding staging resource. How long should the application wait? At least two frames because this will enable parallelism between the CPU(s) and the GPU to be maximally leveraged. The way the GPU works is that while the application is processing frame N by submitting calls to the command buffer, the GPU is busy executing the calls from the previous frame, N-1.

So if an application wants to map a resource that originates in video memory and calls **ID3D10Device::CopyResource** or **ID3D10Device::CopySubresourceRegion** at frame N, this call will actually begin to execute at frame N+1, when the application is submitting

calls for the next frame. The copy should be finished when the application is processing frame N+2.

Frame	GPU/CPU Status
N	• CPU issues render calls for current frame.
N+1	• GPU executing calls sent from CPU during frame N. • CPU issues render calls for current frame.
N+2	• GPU finished executing calls sent from CPU during frame N. Results ready. • GPU executing calls sent from CPU during frame N+1. • CPU issues render calls for current frame.
N+3	• GPU finished executing calls sent from CPU during frame N+1. Results ready. • GPU executing calls sent from CPU during frame N+2. • CPU issues render calls for current frame.
N+4	...

Related topics

Resources (Direct3D 10)

Texture Views (Direct3D 10)

Article • 05/24/2021 • 2 minutes to read

In Direct3D 10, texture resources are accessed with a view, which is a mechanism for hardware interpretation of a resource in memory. A view allows a particular pipeline stage to access only the subresources it needs, in the representation desired by the application.

A view supports the notion of a type-less resource. A type-less resource is a resource created with a specific size but not a specific data type. The data is interpreted dynamically when it is bound to the pipeline.

The following illustration shows an example of binding a 2D texture array with 6 textures as a shader resource by creating a shader resource view for it. The resource is then addressed as an array of textures. (Note: a subresource cannot be bound as both input and output to the pipeline simultaneously.)

When using a 2D texture array as a render target, the resource can be viewed as an array of 2D textures (6 in this example) with mipmap levels (3 in this example).

Create a view object for a render target by calling CreateRenderTargetView. Then call OMSetRenderTargets to set the render target view to the pipeline. Render into the render targets by calling Draw and using the RenderTargetArrayIndex to index into the proper texture in the array. You can use a subresource (a mipmap level, array index combination) to bind to any array of subresources. So you could bind to the second mipmap level and only update this particular mipmap level if you wanted, as in the following illustration.

Differences between Direct3D 9 and Direct3D 10:

- In Direct3D 10, you no longer bind a resource directly to the pipeline, you create a view of a resource, and then set the view to the pipeline. This allows validation and mapping in the runtime and driver to occur at view creation, minimizing type checking at bind-time.

Related topics

Resources (Direct3D 10)

Block Compression (Direct3D 10)

Article • 08/26/2022 • 17 minutes to read

Block compression is a texture compression technique for reducing texture size. When compared to a texture with 32-bits per color, a block-compressed texture can be up to 75 percent smaller. Applications usually see a performance increase when using block compression because of the smaller memory footprint.

While lossy, block compression works well and is recommended for all textures that get transformed and filtered by the pipeline. Textures that are directly mapped to the screen (UI elements like icons and text) are not good choices for compression because artifacts are more noticeable.

A block-compressed texture must be created as a multiple of size 4 in all dimensions and cannot be used as an output of the pipeline.

- How Does Block Compression Work?
  - Storing Uncompressed Data
  - Storing Compressed Data
- Using Block Compression
  - Virtual Size versus Physical Size
- Compression Algorithms
  - BC1
  - BC2
  - BC3
  - BC4
  - BC5
- Format Conversion Using Direct3D 10.1
- Related topics

How Does Block Compression Work?

Block compression is a technique for reducing the amount of memory required to store color data. By storing some colors in their original size, and other colors using an encoding scheme, you can dramatically reduce the amount of memory required to store the image. Since the hardware automatically decodes compressed data, there is no performance penalty for using compressed textures.

To see how compression works, look at the following two examples. The first example describes the amount of memory used when storing uncompressed data; the second example describes the amount of memory used when storing compressed data.

Storing Uncompressed Data

The following illustration represents an uncompressed 4×4 texture. Assume that each color contains a single color component (red for instance) and is stored in one byte of memory.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

The uncompressed data is laid out in memory sequentially and requires 16 bytes, as shown in the following illustration.

1	2	3	...	16

Storing Compressed Data

Now that you've seen how much memory an uncompressed image uses, take a look at how much memory a compressed image saves. The BC4 compression format stores 2 colors (1 byte each) and 16 3-bit indices (48 bits, or 6 bytes) that are used to interpolate the original colors in the texture, as shown in the following illustration.

1	2	16 3-bit encoded colors

The total space required to store the compressed data is 8 bytes which is a 50-percent memory savings over the uncompressed example. The savings are even larger when more than one color component is used.

The substantial memory savings provided by block compression can lead to an increase in performance. This performance comes at the cost of image quality (due to color interpolation); however, the lower quality is often not noticeable.

The next section shows you how Direct3D 10 makes it easy for you to use block compression in your application.

Using Block Compression

Create a block-compressed texture just like an uncompressed texture (see Create a Texture from a File) except that you specify a block-compressed format.

```
loadInfo.Format = DXGI_FORMAT_BC1_UNORM;
D3DX10CreateTextureFromFile(...);
```

Next, create a view to bind the texture to the pipeline. Since a block-compressed texture can be used only as an input to a shader-stage, you want to create a shader-resource view by calling CreateShaderResourceView.

Use a block compressed texture the same way you would use an uncompressed texture. If your application will get a memory pointer to block-compressed data, you need to account for the memory padding in a mipmap that causes the declared size to differ from the actual size.

Virtual Size versus Physical Size

If you have application code that uses a memory pointer to walk the memory of a block compressed texture, there is one important consideration that may require a modification in your application code. A block-compressed texture must be a multiple of 4 in all dimensions because the block-compression algorithms operate on 4x4 texel blocks. This will be a problem for a mipmap whose initial dimensions are divisible by 4, but subdivided levels are not. The following diagram shows the difference in area between the virtual (declared) size and the physical (actual) size of each mipmap level.

The left side of the diagram shows the mipmap level sizes that are generated for an uncompressed 60×40 texture. The top level size is taken from the API call that generates the texture; each subsequent level is half the size of the previous level. For an uncompressed texture, there is no difference between the virtual (declared) size and the physical (actual) size.

The right side of the diagram shows the mipmap level sizes that are generated for the same 60×40 texture with compression. Notice that both the second and third levels have memory padding to make the sizes factors of 4 on every level. This is necessary so that the algorithms can operate on 4×4 texel blocks. This is expecially evident if you consider mipmap levels that are smaller than 4×4; the size of these very small mipmap levels will be rounded up to the nearest factor of 4 when texture memory is allocated.

Sampling hardware uses the virtual size; when the texture is sampled, the memory padding is ignored. For mipmap levels that are smaller than 4×4, only the first four texels will be used for a 2×2 map, and only the first texel will be used by a 1×1 block. However, there is no API structure that exposes the physical size (including the memory padding).

In summary, be careful to use aligned memory blocks when copying regions that contain block-compressed data. To do this in an application that gets a memory pointer, make sure that the pointer uses the surface pitch to account for the physical memory size.

Compression Algorithms

Block compression techniques in Direct3D break up uncompressed texture data into 4×4 blocks, compress each block, and then store the data. For this reason, textures that are expected to be compressed must have texture dimensions that are multiples of 4.

The preceding diagram shows a texture partitioned into texel blocks. The first block shows the layout of the 16 texels labeled a-p, but every block has the same organization of data.

Direct3D implements several compression schemes, each implements a different tradeoff between number of components stored, the number of bits per component, and the amount of memory consumed. Use this table to help choose the format that works best with the type of data and the data resolution that best fits your application.

Source Data	Data Compression Resolution (in bits)	Choose this compression format
Three-component color and alpha	Color (5:6:5), Alpha (1) or no alpha	BC1
Three-component color and alpha	Color (5:6:5), Alpha (4)	BC2
Three-component color and alpha	Color (5:6:5), Alpha (8)	BC3
One-component color	One component (8)	BC4
Two-component color	Two components (8:8)	BC5

BC1

Use the first block compression format (BC1) (either DXGI_FORMAT_BC1_TYPELESS, DXGI_FORMAT_BC1_UNORM, or DXGI_BC1_UNORM_SRGB) to store three-component color data using a 5:6:5 color (5 bits red, 6 bits green, 5 bits blue). This is true even if the data also contains 1-bit alpha. Assuming a 4×4 texture using the largest data format possible, the BC1 format reduces the memory required from 48 bytes (16 colors × 3 components/color × 1 byte/component) to 8 bytes of memory.

The algorithm works on 4×4 blocks of texels. Instead of storing 16 colors, the algorithm saves 2 reference colors (color_0 and color_1) and 16 2-bit color indices (blocks a–p), as shown in the following diagram.

The color indices (a–p) are used to look up the original colors from a color table. The color table contains 4 colors. The first two colors—color_0 and color_1—are the minimum and maximum colors. The other two colors, color_2 and color_3, are intermediate colors calculated with linear interpolation.

```
color_2 = 2/3*color_0 + 1/3*color_1
color_3 = 1/3*color_0 + 2/3*color_1
```

The four colors are assigned 2-bit index values that will be saved in blocks a–p.

```
color_0 = 00
color_1 = 01
color_2 = 10
color_3 = 11
```

Finally, each of the colors in blocks a–p are compared with the four colors in the color table, and the index for the closest color is stored in the 2-bit blocks.

This algorithm lends itself to data that contains 1-bit alpha also. The only difference is that color_3 is set to 0 (which represents a transparent color) and color_2 is a linear blend of color_0 and color_1.

```
color_2 = 1/2*color_0 + 1/2*color_1;
color_3 = 0;
```

Differences between Direct3D 9 and Direct3D 10:

This format exists in both Direct3D 9 and 10.

- In **Direct3D 9**, the BC1 format is called **D3DFMT_DXT1**.
- In **Direct3D 10**, the BC1 format is represented by **DXGI_FORMAT_BC1_UNORM** or **DXGI_FORMAT_BC1_UNORM_SRGB**.

BC2

Use the BC2 format (either DXGI_FORMAT_BC2_TYPELESS, DXGI_FORMAT_BC2_UNORM, or DXGI_BC2_UNORM_SRGB) to store data that contains color and alpha data with low coherency (use BC3 for highly-coherent alpha data). The BC2 format stores RGB data as a 5:6:5 color (5 bits red, 6 bits green, 5 bits blue) and alpha as a separate 4-bit value. Assuming a 4×4 texture using the largest data format possible, this compression technique reduces the memory required from 64 bytes (16 colors × 4 components/color × 1 byte/component) to 16 bytes of memory.

The BC2 format stores colors with the same number of bits and data layout as the BC1 format; however, BC2 requires an additional 64-bits of memory to store the alpha data, as shown in the following diagram.

Differences between Direct3D 9 and Direct3D 10:

This format exists in both Direct3D 9 and 10.

- In **Direct3D 9**, the BC2 format is called **D3DFMT_DXT2** and **D3DFMT_DXT3**.

- In **Direct3D 10**, the BC2 format is represented by **DXGI_FORMAT_BC2_UNORM** or **DXGI_FORMAT_BC2_UNORM_SRGB**

BC3

Use the BC3 format (either DXGI_FORMAT_BC3_TYPELESS, DXGI_FORMAT_BC3_UNORM, or DXGI_BC3_UNORM_SRGB) to store highly coherent color data (use BC2 with less coherent alpha data). The BC3 format stores color data using 5:6:5 color (5 bits red, 6 bits green, 5 bits blue) and alpha data using one byte. Assuming a 4×4 texture using the largest data format possible, this compression technique reduces the memory required from 64 bytes (16 colors × 4 components/color × 1 byte/component) to 16 bytes of memory.

The BC3 format stores colors with the same number of bits and data layout as the BC1 format; however, BC3 requires an additional 64-bits of memory to store the alpha data. The BC3 format handles alpha by storing two reference values and interpolating between them (similarly to how BC1 stores RGB color).

The algorithm works on 4×4 blocks of texels. Instead of storing 16 alpha values, the algorithm stores 2 reference alphas (alpha_0 and alpha_1) and 16 3-bit color indices (alpha a through p), as shown in the following diagram.

The BC3 format uses the alpha indices (a–p) to look up the original colors from a lookup table that contains 8 values. The first two values—alpha_0 and alpha_1—are the

minimum and maximum values; the other six intermediate values are calculated using linear interpolation.

The algorithm determines the number of interpolated alpha values by examining the two reference alpha values. If alpha_0 is greater than alpha_1, then BC3 interpolates 6 alpha values; otherwise, it interpolates 4. When BC3 interpolates only 4 alpha values, it sets two additional alpha values (0 for fully transparent and 255 for fully opaque). BC3 compresses the alpha values in the 4×4 texel area by storing the bit code corresponding to the interpolated alpha values which most closely matches the original alpha for a given texel.

```
if( alpha_0 > alpha_1 )
{
  // 6 interpolated alpha values.
  alpha_2 = 6/7*alpha_0 + 1/7*alpha_1; // bit code 010
  alpha_3 = 5/7*alpha_0 + 2/7*alpha_1; // bit code 011
  alpha_4 = 4/7*alpha_0 + 3/7*alpha_1; // bit code 100
  alpha_5 = 3/7*alpha_0 + 4/7*alpha_1; // bit code 101
  alpha_6 = 2/7*alpha_0 + 5/7*alpha_1; // bit code 110
  alpha_7 = 1/7*alpha_0 + 6/7*alpha_1; // bit code 111
}
else
{
  // 4 interpolated alpha values.
  alpha_2 = 4/5*alpha_0 + 1/5*alpha_1; // bit code 010
  alpha_3 = 3/5*alpha_0 + 2/5*alpha_1; // bit code 011
  alpha_4 = 2/5*alpha_0 + 3/5*alpha_1; // bit code 100
  alpha_5 = 1/5*alpha_0 + 4/5*alpha_1; // bit code 101
  alpha_6 = 0;                         // bit code 110
  alpha_7 = 255;                       // bit code 111
}
```

Differences between Direct3D 9 and Direct3D 10:

- In **Direct3D 9**, the BC3 format is called **D3DFMT_DXT4** and **D3DFMT_DXT5**.
- In **Direct3D 10**, the BC3 format is represented by **DXGI_FORMAT_BC3_UNORM** or **DXGI_FORMAT_BC3_UNORM_SRGB**.

BC4

Use the BC4 format to store one-component color data using 8 bits for each color. As a result of the increased accuracy (compared to BC1), BC4 is ideal for storing floating-point data in the range of [0 to 1] using the DXGI_FORMAT_BC4_UNORM format and [-1 to +1] using the DXGI_FORMAT_BC4_SNORM format. Assuming a 4×4 texture using the

largest data format possible, this compression technique reduces the memory required from 16 bytes (16 colors × 1 components/color × 1 byte/component) to 8 bytes.

The algorithm works on 4×4 blocks of texels. Instead of storing 16 colors, the algorithm stores 2 reference colors (red_0 and red_1) and 16 3-bit color indices (red a through red p), as shown in the following diagram.

The algorithm uses the 3-bit indices to look up colors from a color table that contains 8 colors. The first two colors—red_0 and red_1—are the minimum and maximum colors. The algorithm calculates the remaining colors using linear interpolation.

The algorithm determines the number of interpolated color values by examining the two reference values. If red_0 is greater than red_1, then BC4 interpolates 6 color values; otherwise, it interpolates 4. When BC4 interpolates only 4 color values, it sets two additional color values (0.0f for fully transparent and 1.0f for fully opaque). BC4 compresses the alpha values in the 4×4 texel area by storing the bit code corresponding to the interpolated alpha values that most closely matches the original alpha for a given texel.

- BC4_UNORM
- BC4_SNORM

BC4_UNORM

The interpolation of the single-component data is done as in the following code sample.

```
unsigned word red_0, red_1;

if( red_0 > red_1 )
{
  // 6 interpolated color values
  red_2 = (6*red_0 + 1*red_1)/7.0f; // bit code 010
  red_3 = (5*red_0 + 2*red_1)/7.0f; // bit code 011
  red_4 = (4*red_0 + 3*red_1)/7.0f; // bit code 100
  red_5 = (3*red_0 + 4*red_1)/7.0f; // bit code 101
```

```
  red_6 = (2*red_0 + 5*red_1)/7.0f; // bit code 110
  red_7 = (1*red_0 + 6*red_1)/7.0f; // bit code 111
}
else
{
  // 4 interpolated color values
  red_2 = (4*red_0 + 1*red_1)/5.0f; // bit code 010
  red_3 = (3*red_0 + 2*red_1)/5.0f; // bit code 011
  red_4 = (2*red_0 + 3*red_1)/5.0f; // bit code 100
  red_5 = (1*red_0 + 4*red_1)/5.0f; // bit code 101
  red_6 = 0.0f;                     // bit code 110
  red_7 = 1.0f;                     // bit code 111
}
```

The reference colors are assigned 3-bit indices (000–111 since there are 8 values), which will be saved in blocks red a through red p during compression.

BC4_SNORM

The DXGI_FORMAT_BC4_SNORM is exactly the same, except that the data is encoded in SNORM range and when 4 color values are interpolated. The interpolation of the single-component data is done as in the following code sample.

```
signed word red_0, red_1;

if( red_0 > red_1 )
{
  // 6 interpolated color values
  red_2 = (6*red_0 + 1*red_1)/7.0f; // bit code 010
  red_3 = (5*red_0 + 2*red_1)/7.0f; // bit code 011
  red_4 = (4*red_0 + 3*red_1)/7.0f; // bit code 100
  red_5 = (3*red_0 + 4*red_1)/7.0f; // bit code 101
  red_6 = (2*red_0 + 5*red_1)/7.0f; // bit code 110
  red_7 = (1*red_0 + 6*red_1)/7.0f; // bit code 111
}
else
{
  // 4 interpolated color values
  red_2 = (4*red_0 + 1*red_1)/5.0f; // bit code 010
  red_3 = (3*red_0 + 2*red_1)/5.0f; // bit code 011
  red_4 = (2*red_0 + 3*red_1)/5.0f; // bit code 100
  red_5 = (1*red_0 + 4*red_1)/5.0f; // bit code 101
  red_6 = -1.0f;                     // bit code 110
  red_7 =  1.0f;                     // bit code 111
}
```

The reference colors are assigned 3-bit indices (000–111 since there are 8 values), which will be saved in blocks red a through red p during compression.

BC5

Use the BC5 format to store two-component color data using 8 bits for each color. As a result of the increased accuracy (compared to BC1), BC5 is ideal for storing floating-point data in the range of [0 to 1] using the DXGI_FORMAT_BC5_UNORM format and [-1 to +1] using the DXGI_FORMAT_BC5_SNORM format. Assuming a 4×4 texture using the largest data format possible, this compression technique reduces the memory required from 32 bytes (16 colors × 2 components/color × 1 byte/component) to 16 bytes.

The algorithm works on 4×4 blocks of texels. Instead of storing 16 colors for both components, the algorithm stores 2 reference colors for each component (red_0, red_1, green_0, and green_1) and 16 3-bit color indices for each component (red a through red p and green a through green p), as shown in the following diagram.

The algorithm uses the 3-bit indices to look up colors from a color table that contains 8 colors. The first two colors—red_0 and red_1 (or green_0 and green_1)—are the minimum and maximum colors. The algorithm calculates the remaining colors using linear interpolation.

The algorithm determines the number of interpolated color values by examining the two reference values. If red_0 is greater than red_1, then BC5 interpolates 6 color values; otherwise, it interpolates 4. When BC5 interpolates only 4 color values, it sets the remaining two color values at 0.0f and 1.0f.

- BC5_UNORM

- BC5_SNORM

BC5_UNORM

The interpolation of the single-component data is done as in the following code sample. The calculations for the green components are similar.

```
unsigned word red_0, red_1;

if( red_0 > red_1 )
{
  // 6 interpolated color values
  red_2 = (6*red_0 + 1*red_1)/7.0f; // bit code 010
  red_3 = (5*red_0 + 2*red_1)/7.0f; // bit code 011
  red_4 = (4*red_0 + 3*red_1)/7.0f; // bit code 100
  red_5 = (3*red_0 + 4*red_1)/7.0f; // bit code 101
  red_6 = (2*red_0 + 5*red_1)/7.0f; // bit code 110
  red_7 = (1*red_0 + 6*red_1)/7.0f; // bit code 111
}
else
{
  // 4 interpolated color values
  red_2 = (4*red_0 + 1*red_1)/5.0f; // bit code 010
  red_3 = (3*red_0 + 2*red_1)/5.0f; // bit code 011
  red_4 = (2*red_0 + 3*red_1)/5.0f; // bit code 100
  red_5 = (1*red_0 + 4*red_1)/5.0f; // bit code 101
  red_6 = 0.0f;                     // bit code 110
  red_7 = 1.0f;                     // bit code 111
}
```

The reference colors are assigned 3-bit indices (000–111 since there are 8 values), which will be saved in blocks red a through red p during compression.

BC5_SNORM

The DXGI_FORMAT_BC5_SNORM is exactly the same, except that the data is encoded in SNORM range and when 4 data values are interpolated, the two additional values are -1.0f and 1.0f. The interpolation of the single-component data is done as in the following code sample. The calculations for the green components are similar.

```
signed word red_0, red_1;

if( red_0 > red_1 )
{
```

```
    // 6 interpolated color values
    red_2 = (6*red_0 + 1*red_1)/7.0f; // bit code 010
    red_3 = (5*red_0 + 2*red_1)/7.0f; // bit code 011
    red_4 = (4*red_0 + 3*red_1)/7.0f; // bit code 100
    red_5 = (3*red_0 + 4*red_1)/7.0f; // bit code 101
    red_6 = (2*red_0 + 5*red_1)/7.0f; // bit code 110
    red_7 = (1*red_0 + 6*red_1)/7.0f; // bit code 111
  }
  else
  {
    // 4 interpolated color values
    red_2 = (4*red_0 + 1*red_1)/5.0f; // bit code 010
    red_3 = (3*red_0 + 2*red_1)/5.0f; // bit code 011
    red_4 = (2*red_0 + 3*red_1)/5.0f; // bit code 100
    red_5 = (1*red_0 + 4*red_1)/5.0f; // bit code 101
    red_6 = -1.0f;                    // bit code 110
    red_7 =  1.0f;                    // bit code 111
  }
```

The reference colors are assigned 3-bit indices (000–111 since there are 8 values), which will be saved in blocks red a through red p during compression.

Format Conversion Using Direct3D 10.1

Direct3D 10.1 enables copies between prestructured-typed textures and block-compressed textures of the same bit widths. The functions that can accomplish this are CopyResource and CopySubresourceRegion.

Beginning with Direct3D 10.1, you can use CopyResource and CopySubresourceRegion to copy between a few format types. This type of copy operation performs a type of format conversion that reinterprets resource data as a different format type. Consider this example that shows the difference between reinterpreting data with the way a more typical type of conversion behaves:

```
FLOAT32 f = 1.0f;
UINT32 u;
```

To reinterpret ‘f’ as the type of ‘u’, use memcpy:

```
memcpy( &u, &f, sizeof( f ) ); // ‘u’ becomes equal to 0x3F800000.
```

In the preceding reinterpretation, the underlying value of the data doesn't change; memcpy reinterprets the float as an unsigned integer.

To perform the more typical type of conversion, use assignment:

```
u = f; // 'u' becomes 1.
```

In the preceding conversion, the underlying value of the data changes.

The following table lists the allowable source and destination formats that you can use in this reinterpretation type of format conversion. You must encode the values properly for the reinterpretation to work as expected.

Bit Width	Uncompressed Resource	Block-Compressed Resource
32	DXGI_FORMAT_R32_UINT DXGI_FORMAT_R32_SINT	DXGI_FORMAT_R9G9B9E5_SHAREDEXP
64	DXGI_FORMAT_R16G16B16A16_UINT DXGI_FORMAT_R16G16B16A16_SINT DXGI_FORMAT_R32G32_UINT DXGI_FORMAT_R32G32_SINT	DXGI_FORMAT_BC1_UNORM[_SRGB] DXGI_FORMAT_BC4_UNORM DXGI_FORMAT_BC4_SNORM
128	DXGI_FORMAT_R32G32B32A32_UINT DXGI_FORMAT_R32G32B32A32_SINT	DXGI_FORMAT_BC2_UNORM[_SRGB] DXGI_FORMAT_BC3_UNORM[_SRGB] DXGI_FORMAT_BC5_UNORM DXGI_FORMAT_BC5_SNORM

Related topics

Resources (Direct3D 10)

Resource Limits (Direct3D 10)

Article • 01/06/2021 • 2 minutes to read

This table contains a list of the minimum resources supported by Direct3D 10.

Resource	Limit
Number of elements in a constant buffer	4096
Number of texels (independent of struct size) in a buffer	227 texels
Texture1D U dimension	8192
Texture1DArray dimension	512 Array Slices
Texture2D U/V dimension	8192
Texture2DArray dimension	512 Array Slices
Texture3D U/V/W dimension	2048
TextureCube dimension	8192
Resource size (in MB)	128 MB[1]
Anisotropic filtering maxanisotropy	16
Resource dimension addressable by filtering hardware	8192 per dimension
Resource size (in MB) addressable by IA (input or vertex data) or VS/GS/PS (point sample)	128 MB[1]
Total number of resource views per context (Each array counts as 1) (all view types have shared limit)	220
Buffer structure size (multi-element)	2048 bytes
Stream output size	Same as the number of texels in a buffer (see above)
Draw or DrawInstanced vertex count (including instancing)	232
DrawIndexed[Instanced]() vertex count (incl. instancing)	232
GS invocation output data (components * vertices)	1024
Total number of sampler objects per context	4096
Total number of viewport/scissor objects per pipeline	16
Total number of clip/cull distances per vertex	8

Resource	Limit
Total number of blend objects per context	4096
Total number of depth/stencil objects per context	4096
Total number of rasterizer state objects per context	4096
Maximum per-pixel sample count during multisampling	32
Shader resource vertex-element count (four 32-bit components)	16
Common-shader core (four 32-bit components) temp-register count (r# + indexable x#[n])	4096
Common-shader core constant-buffer slots	14
Common-shader core input-resource slots	128
Common-shader core sampler slots	16
Common-shader core subroutine-nesting limit	32
Common-shader core flow-control nesting limit	64
Vertex shader input-register count (four 32-bit components)	16
Vertex shader output-register count (four 32-bit components)	16
Geometry shader input-register count (four 32-bit components)	16
Geometry shader output-register count (four 32-bit components)	32
Pixel shader input-register count (four 32-bit components)	32
Pixel shader output-register count (four 32-bit components)	8
Pixel shader output depth register count(32-bit*1-component)	1
Input assembler index input resource slots	1
Input assembler vertex input resource slots	16

[1]Apps can create resources bigger than the maximum resource size on some graphics hardware. However, we recommend that apps keep resources smaller than the maximum resource size to get the maximum amount of compatibility across graphics vendors. The runtime only guarantees that allocations within the maximum resource size

are supported by all Direct3D 10 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.

Related topics

Resources (Direct3D 10)

Coordinate Systems (Direct3D 10)

Article • 05/24/2021 • 2 minutes to read

Coordinate systems for Direct3D 10 are defined for pixels and texels.

Differences between Direct3D 9 and Direct3D 10:

- Direct3D 10 defines the upper-left corner of the upper-left pixel as the origin of a render target.
- Direct3D 9 defines the center of the upper-left pixel as the origin of a render target.

Pixel Coordinate System

- Pixel Coordinate System for Direct3D 9

Texel Coordinate System

- Texel Coordinate System

Related topics

Pixel Coordinate System

The pixel coordinate system in Direct3D 10 defines the origin of a render target at the upper-left corner. as shown in the following diagram. Pixel centers are offset by (0.5f,0.5f) from integer locations.

Pixel Coordinate System for Direct3D 9

For reference, here is the pixel coordinate system for Direct3D 9, which defined the origin or a render target as the center of the upper-left pixel, (0.5,0.5) away from the upper left corner, as shown in the following diagram. In Direct3D 9, pixel centers are at integer locations.

Texel Coordinate System

The texel coordinate system has its origin at the top-left corner of the texture, as shown in the following diagram. This makes rendering screen-aligned textures trivial (in Direct3D 10), as the pixel coordinate system is aligned with the texel coordinate system.

Texel Coordinate System

Texture coordinates are represented with either a normalized or a scaled number; each texture coordinate is mapped to a specific texel as follows:

For a normalized coordinate:

- Point sampling: Texel # = floor(U * Width)

- Linear sampling: Left Texel # = floor(U * Width), Right Texel # = Left Texel # + 1

For a scaled coordinate:

- Point sampling: Texel # = floor(U)
- Linear sampling: Left Texel # = floor(U - 0.5), Right Texel # = Left Texel # + 1

Where the width, is the width of the texture (in texels).

Texture address wrapping occurs after the texel location is computed.

Related topics

Resources (Direct3D 10)

Floating-point rules (Direct3D 10)

Article • 01/06/2021 • 5 minutes to read

Direct3D 10 supports several different floating-point representations. All floating-point computations operate under a defined subset of the IEEE 754 32-bit single precision floating-point behavior.

- 32-bit Floating-Point Rules
  - Honored IEEE-754 Rules
  - Deviations or Additional Requirements from IEEE-754 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 10, however, defines a looser requirement: 32-bit floating-point operations produce a result that is within one unit-last-place (1 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 one ULP.
- 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 does not manipulate the data.
- States that contain floating-point values, such as Viewport MinDepth/MaxDepth, BorderColor values etc., may be provided as denorm values and may or may not be flushed before use by the hardware.
- 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, however the exact bit pattern of the NaN is not required to stay the same (unless the operation is a raw move instruction - which does not alter data at all.)
- Min or max operations for which only one operand is NaN return the other operand as the result (contrary to comparison rules above). This is a new IEEE rule (IEEE 754R), required in Direct3D 10.
- Another new IEEE 754R rule is that min(-0,+0) == min(+0,-0) == -0, and max(-0,+0) == max(+0,-0) == +0, which honor the sign, in contrast to the comparison rules for signed zero (stated above). Direct3D 10 recommends the IEEE 754R behavior here, but it will not be enforced; 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. Note that 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 10 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.

- Multiply and divide each operate at the 32-bit floating-point precision level (accuracy to 1 ULP).

16-bit Floating-Point Rules

Direct3D 10 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 the following rules:

- if e == 31 and f != 0, then v is NaN regardless of s
- if e == 31 and f == 0, then v = (-1)s*infinity (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 above. 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 10 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 = +infinity
- 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 above. 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 (Direct3D 10)

Data Conversion Rules

Article • 08/20/2021 • 11 minutes to read

The following sections describe how Direct3D handles conversions between data types.

- Data Type Terminology
- Floating Point Conversion
  - Conververting from a higher range representation to a lower range representation
  - Converting from a lower range representation to a higher range representation
- Integer Conversion
- Fixed Point Integer Conversion
- Related topics

Data Type Terminology

The following set of terms are subsequently used to characterize various format conversions.

Term	Definition
SNORM	Signed normalized integer, meaning that for an n-bit 2's complement number, the maximum value means 1.0f (e.g. the 5-bit value 01111 maps to 1.0f), and the minimum value means -1.0f (e.g. the 5-bit value 10000 maps to -1.0f). In addition, the second-minimum number maps to -1.0f (e.g. the 5-bit value 10001 maps to -1.0f). There are thus two integer representations for -1.0f. There is a single representation for 0.0f, and a single representation for 1.0f. This results in a set of integer representations for evenly spaced floating point values in the range (-1.0f...0.0f), and also a complementary set of representations for numbers in the range (0.0f...1.0f)
UNORM	Unsigned normalized integer, meaning that for an n-bit number, all 0's means 0.0f, and all 1's means 1.0f. A sequence of evenly spaced floating point values from 0.0f to 1.0f are represented. e.g. a 2-bit UNORM represents 0.0f, 1/3, 2/3, and 1.0f.
SINT	Signed integer. 2's complement integer. e.g. an 3-bit SINT represents the integral values -4, -3, -2, -1, 0, 1, 2, 3.
UINT	Unsigned integer. e.g. a 3-bit UINT represents the integral values 0, 1, 2, 3, 4, 5, 6, 7.
FLOAT	A floating-point value in any of the representations defined by Direct3D.
SRGB	Similar to UNORM, in that for an n-bit number, all 0's means 0.0f and all 1's means 1.0f. However unlike UNORM, with SRGB the sequence of unsigned integer encodings between all 0's to all 1's represent a nonlinear progression in the floating point interpretation of the numbers, between 0.0f to 1.0f. Roughly, if this nonlinear progression, SRGB, is displayed as a sequence of colors, it would appear as a linear ramp of luminosity levels to an "average" observer, under "average" viewing conditions, on an "average" display. For complete detail, refer to the SRGB color standard, IEC 61996-2-1, at IEC (International Electrotechnical Commission).

Floating Point Conversion

Whenever a floating point conversion between different representations occurs, including to or from non-floating point representations, the following rules apply.

Conververting from a higher range representation to a lower range representation

- Round-to-zero is used during conversion to another float format. If the target is an integer or fixed point format, round-to-nearest-even is used, unless the conversion is explicitly documented as using another rounding behavior, such as round-to-nearest for FLOAT to SNORM, FLOAT to UNORM or FLOAT to SRGB. Other exceptions are the ftoi and ftou shader instructions, which use round-to-zero. Finally, the float-to-fixed conversions used by the texture sampler and rasterizer have a specified tolerance measured in Unit-Last-Place from an infinitely precise ideal.

- For source values greater than the dynamic range of a lower range target format (eg. a large 32-bit float value is written into a 16-bit float RenderTarget), the maximum representable (appropriately signed) value results, NOT including signed infinity (due to the round to zero described above).
- NaN in a higher range format will be converted to NaN representation in the lower range format if the NaN representation exists in the lower range format. If the lower format does not have a NaN representation, the result will be 0.
- INF in a higher range format will be converted to INF in the lower range format if available. If the lower format does not have an INF representation, it will be converted to the maximum value representable. The sign will be preserved if available in the target format.
- Denorm in a higher range format will be converted to the Denorm representation in the lower range format if available in the lower range format and the conversion is possible, otherwise the result is 0. The sign bit will be preserved if available in the target format.

Converting from a lower range representation to a higher range representation

- NaN in a lower range format will be converted to the NaN representation in the higher range format if available in the higher range format. If the higher range format does not have a NaN representation, it will be converted to 0.
- INF in a lower range format will be converted to the INF representation in the higher range format if available in the higher range format. If the higher format does not have an INF representation, it will be converted to the maximum value representable (MAX_FLOAT in that format). The sign will be preserved if available in the target format.
- Denorm in a lower range format will be converted to a normalized representation in the higher range format if possible, or else to a Denorm representation in the higher range format if the Denorm representation exists. Failing those, if the higher range format does not have a Denorm representation, it will be converted to 0. The sign will be preserved if available in the target format. Note that 32-bit float numbers count as a format without a Denorm representation (because Denorms in operations on 32-bit floats flush to sign preserved 0).

Integer Conversion

The following table describes conversions from various representations described above to other representations. Only conversions that actually occur in Direct3D are shown.

Source Data Type	Destination Data Type	Conversion Rule
SNORM	FLOAT	Given an n-bit integer value representing the signed range [-1.0f to 1.0f], conversion to floating-point is as follows. • The most-negative value maps to -1.0f. e.g. the 5-bit value 10000 maps to -1.0f. • Every other value is converted to a float (call it c), and then result = c * (1.0f / ($2^{(n-1)}$-1)). For example the 5-bit value 10001 is converted to -15.0f, and then divided by 15.0f, yielding -1.0f.

FLOAT	SNORM	Given a floating-point number, conversion to an n-bit integer value representing the signed range [-1.0f to 1.0f] is as follows.
		• Let c represent the starting value.
		• If c is NaN, the result is 0.
		• If c > 1.0f, including INF, it is clamped to 1.0f.
		• If c < -1.0f, including -INF, it is clamped to -1.0f.
		• Convert from float scale to integer scale: c = c * (2^{n-1}-1).
		• Convert to an integer as follows.
		  ◦ If c >= 0 then c = c + 0.5f, otherwise, c = c - 0.5f.
		  ◦ Drop the decimal fraction, and the remaining floating point (integral) value is converted directly to an integer.
		This conversion is permitted a tolerance of D3D*xx*_FLOAT32_TO_INTEGER_TOLERANCE_IN_Unit-Last-Place Unit-Last-Place (on the integer side). This means that after converting from float to integer scale, any value within D3D*xx*_FLOAT32_TO_INTEGER_TOLERANCE_IN_ULP Unit-Last-Place of a representable target format value is permitted to map to that value. The additional Data Invertability requirement ensures that the conversion is nondecreasing across the range and all output values are attainable. (In the constants shown here, *xx* should be replaced with the Direct3D version, for example 10, 11, or 12.)

Source Data Type	Destination Data Type	Conversion Rule
UNORM	FLOAT	The starting n-bit value is converted to float (0.0f, 1.0f, 2.0f, etc.) and then divided by (2^n-1).

FLOAT	UNORM	Let c represent the starting value.
		
		• If c is NaN, the result is 0.
		• If c > 1.0f, including INF, it is clamped to 1.0f.
		• If c < 0.0f, including -INF, it is clamped to 0.0f.
		• Convert from float scale to integer scale: c = c * (2^n-1).
		• Convert to integer.
		  ○ c = c + 0.5f.
		  ○ The decimal fraction is dropped, and the remaining floating point (integral) value is converted directly to an integer.
		
		This conversion is permitted a tolerance of D3D*xx*_FLOAT32_TO_INTEGER_TOLERANCE_IN_ULP Unit-Last-Place (on the integer side). This means that after converting from float to integer scale, any value within D3D*xx*_FLOAT32_TO_INTEGER_TOLERANCE_IN_ULP Unit-Last-Place of a representable target format value is permitted to map to that value. The additional Data Invertability requirement ensures that the conversion is nondecreasing across the range and all output values are attainable.

SRGB	FLOAT	The following is the ideal SRGB to FLOAT conversion. • Take the starting n-bit value, convert it a float (0.0f, 1.0f, 2.0f, etc.); call this c. • c = c * (1.0f / (2^n-1)) • If (c < = D3D*xx*_SRGB_TO_FLOAT_THRESHOLD) then: result = c / D3D*xx*_SRGB_TO_FLOAT_DENOMINATOR_1, else: result = ((c + D3D*xx*_SRGB_TO_FLOAT_OFFSET)/D3D*xx*_SRGB_TO_FLOAT_DENOMINATOR_2)D3D*xx*_SRGB_TO_FLOAT_EXPONENT This conversion is permitted a tolerance of D3D*xx*_SRGB_TO_FLOAT_TOLERANCE_IN_ULP Unit-Last-Place (on the SRGB side).

FLOAT	SRGB	The following is the ideal FLOAT -> SRGB conversion.
		Assuming the target SRGB color component has n bits:
		
		• Suppose the starting value is c.
		• If c is NaN, the result is 0.
		• If c > 1.0f, including INF, is clamped to 1.0f.
		• If c < 0.0f, including -INF, it is clamped to 0.0f.
		• If (c <= D3D*xx*_FLOAT_TO_SRGB_THRESHOLD) then: c = D3D*xx*_FLOAT_TO_SRGB_SCALE_1 * c, else: c = D3D*xx*_FLOAT_TO_SRGB_SCALE_2 * c(D3D*xx*_FLOAT_TO_SRGB_EXPONENT_NUMERATOR/D3D*xx*_FLOAT_TO_SRGB_EXPONENT_DENOMINATOR) - D3D*xx*_FLOAT_TO_SRGB_OFFSET
		• Convert from float scale to integer scale: c = c * (2^n-1).
		• Convert to integer:
		  ○ c = c + 0.5f.
		  ○ The decimal fraction is dropped, and the remaining floating point (integral) value is converted directly to an integer.
		
		This conversion is permitted a tolerance of D3D*xx*_FLOAT32_TO_INTEGER_TOLERANCE_IN_ULP Unit-Last-Place (on the integer side). This means that after converting from float to integer scale, any value within D3D*xx*_FLOAT32_TO_INTEGER_TOLERANCE_IN_ULP Unit-Last-Place of a representable target format value is permitted to map to that value. The additional Data Invertability requirement ensures that the conversion is nondecreasing across the range and all output values are attainable.

SINT	SINT With More Bits	To convert from SINT to an SINT with more bits, the most significant bit (MSB) of the starting number is "sign-extended" to the additional bits available in the target format.
UINT	SINT With More Bits	To convert from UINT to an SINT with more bits, the number is copied to the target format's least significant bits (LSBs) and additional MSBs are padded with 0.
SINT	UINT With More Bits	To convert from SINT to UINT with more bits: If negative, the value is clamped to 0. Otherwise the number is copied to the target format's LSBs and additional MSB's are padded with 0.
UINT	UINT With More Bits	To convert from UINT to UINT with more bits the number is copied to the target format's LSBs and additional MSB's are padded with 0.

Source Data Type	Destination Data Type	Conversion Rule
SINT or UINT	SINT or UINT With Fewer or Equal Bits	To convert from a SINT or UINT to SINT or UINT with fewer or equal bits (and/or change in signedness), the starting value is simply clamped to the range of the target format.

Fixed Point Integer Conversion

Fixed point integers are simply integers of some bit size that have an implicit decimal point at a fixed location.

The ubiquitous "integer" data type is a special case of a fixed point integer with the decimal at the end of the number.

Fixed point number representations are characterized as: i.f, where i is the number of integer bits and f is the number of fractional bits. e.g. 16.8 means 16 bits integer followed by 8 bits of fraction. The integer part is stored in 2's complement, at least as defined here (though it can be defined equally for unsigned integers as well). The fractional part is stored in unsigned form. The fractional part always represents the positive fraction between the two nearest integral values, starting from the most negative.

Addition and subtraction operations on fixed point numbers are performed simply using standard integer arithmetic, without any consideration for where the implied decimal lies. Adding 1 to a 16.8 fixed point number just means adding 256, since the decimal is 8 places in from the least significant end of the number. Other operations such as multiplication, can be performed as well simply using integer arithmetic, provided the effect on the fixed decimal is accounted for. For example, multiplying two 16.8 integers using an integer multiply produces a 32.16 result.

Fixed point integer representations are used in two ways in Direct3D.

- Post-clipped vertex positions in the rasterizer are snapped to fixed point, to uniformly distribute precision across the RenderTarget area. Many rasterizer operations, including face culling as one example, occur on fixed point snapped positions, while other operations, such as attribute interpolator setup, use positions that have been converted back to floating point from the fixed point snapped positions.
- Texture coordinates for sampling operations are snapped to fixed point (after being scaled by texture size), to uniformly distribute precision across texture space, in choosing filter tap locations/weights. Weight values are converted back to floating point before actual filtering arithmetic is performed.

Source Data Type	Destination Data Type	Conversion Rule
FLOAT	Fixed Point Integer	The following is the general procedure for converting a floating point number n to a fixed point integer i.f, where i is the number of (signed) integer bits and f is the number of fractional bits. • Compute FixedMin = $-2^{(i-1)}$ • Compute FixedMax = $2^{(i-1)} - 2^{(-f)}$ • If n is a NaN, result = 0; if n is +Inf, result = FixedMax*2^f; if n is -Inf, result = FixedMin*2^f • If n >= FixedMax, result = Fixedmax*2^f; if n <= FixedMin, result = FixedMin*2^f • Else compute n*2^f and convert to integer. Implementations are permitted D3D*xx*_FLOAT32_TO_INTEGER_TOLERANCE_IN_ULP Unit-Last-Place tolerance in the integer result, instead of the infinitely precise value n*2^f after the last step above.
Fixed Point Integer	FLOAT	Assume that the specific fixed point representation being converted to float does not contain more than a total of 24 bits of information, no more than 23 bits of which is in the fractional component. Suppose a given fixed point number, fxp, is in i.f form (i bits integer, f bits fraction). The conversion to float is akin to the following pseudocode. float result = (float)(fxp >> f) + // extract integer ((float)(fxp & (2^f - 1)) / (2^f)); // extract fraction

Related topics

Resources (Direct3D 10)

Legacy Formats: Map Direct3D 9 Formats to Direct3D 10

Article • 01/06/2021 • 2 minutes to read

This table lists the Direct3D 9 formats that can be mapped to a Direct3D 10 format. The Direct3D 10 formats have diverged from the Direct3D 9 formats so that vertex and texture formats can be merged to enable cross-endian applications.

Direct3D 9 Texture/Vertex/Index Format	**Equivalent Direct3D 10 Format**
D3DFMT_UNKNOWN	DXGI_FORMAT_UNKNOWN
D3DFMT_R8G8B8	Not available
D3DFMT_A8R8G8B8	DXGI_FORMAT_B8G8R8A8_UNORM (DXGI 1.1)
D3DFMT_X8R8G8B8	DXGI_FORMAT_B8G8R8X8_UNORM (DXGI 1.1)
D3DFMT_R5G6B5	DXGI_FORMAT_B5G6R5_UNORM (DXGI 1.2)
D3DFMT_X1R5G5B5	Not available
D3DFMT_A1R5G5B5	DXGI_FORMAT_B5G5R5A1_UNORM (DXGI 1.2)
D3DFMT_A4R4G4B4	DXGI_FORMAT_B4G4R4A4_UNORM (DXGI 1.2)
D3DFMT_R3G3B2	Not available
D3DFMT_A8	DXGI_FORMAT_A8_UNORM
D3DFMT_A8R3G3B2	Not available
D3DFMT_X4R4G4B4	Not available
D3DFMT_A2B10G10R10	DXGI_FORMAT_R10G10B10A2
D3DFMT_A8B8G8R8	DXGI_FORMAT_R8G8B8A8_UNORM or DXGI_FORMAT_R8G8B8A8_UNORM_SRGB
D3DFMT_X8B8G8R8	Not available
D3DFMT_G16R16	DXGI_FORMAT_R16G16_UNORM
D3DFMT_A2R10G10B10	Not available
D3DFMT_A16B16G16R16	DXGI_FORMAT_R16G16B16A16_UNORM
D3DFMT_A8P8	Not available

Direct3D 9 Texture/Vertex/Index Format	Equivalent Direct3D 10 Format
D3DFMT_P8	Not available
D3DFMT_L8	DXGI_FORMAT_R8_UNORM [1]
D3DFMT_A8L8	Not available
D3DFMT_A4L4	Not available
D3DFMT_V8U8	DXGI_FORMAT_R8G8_SNORM
D3DFMT_L6V5U5	Not available
D3DFMT_X8L8V8U8	Not available
D3DFMT_Q8W8V8U8	DXGI_FORMAT_R8G8B8A8_SNORM
D3DFMT_V16U16	DXGI_FORMAT_R16G16_SNORM
D3DFMT_W11V11U10	Not available
D3DFMT_A2W10V10U10	Not available
D3DFMT_UYVY	Not available
D3DFMT_R8G8_B8G8	DXGI_FORMAT_G8R8_G8B8_UNORM [2]
D3DFMT_YUY2	Not available
D3DFMT_G8R8_G8B8	DXGI_FORMAT_R8G8_B8G8_UNORM [2]
D3DFMT_DXT1	DXGI_FORMAT_BC1_UNORM or DXGI_FORMAT_BC1_UNORM_SRGB
D3DFMT_DXT2	DXGI_FORMAT_BC2_UNORM or DXGI_FORMAT_BC2_UNORM_SRGB [3]
D3DFMT_DXT3	DXGI_FORMAT_BC2_UNORM or DXGI_FORMAT_BC2_UNORM_SRGB
D3DFMT_DXT4	DXGI_FORMAT_BC3_UNORM or DXGI_FORMAT_BC3_UNORM_SRGB [3]
D3DFMT_DXT5	DXGI_FORMAT_BC3_UNORM or DXGI_FORMAT_BC3_UNORM_SRGB
D3DFMT_D16 and D3DFMT_D16_LOCKABLE	DXGI_FORMAT_D16_UNORM
D3DFMT_D32	Not available

Direct3D 9 Texture/Vertex/Index Format	Equivalent Direct3D 10 Format
D3DFMT_D15S1	Not available
D3DFMT_D24S8	Not available
D3DFMT_D24X8	Not available
D3DFMT_D24X4S4	Not available
D3DFMT_D16	DXGI_FORMAT_D16_UNORM
D3DFMT_D32F_LOCKABLE	DXGI_FORMAT_D32_FLOAT
D3DFMT_D24FS8	Not available
D3DFMT_S1D15	Not available
D3DFMT_S8D24	DXGI_FORMAT_D24_UNORM_S8_UINT
D3DFMT_X8D24	Not available
D3DFMT_X4S4D24	Not available
D3DFMT_L16	DXGI_FORMAT_R16_UNORM [1]
D3DFMT_INDEX16	DXGI_FORMAT_R16_UINT
D3DFMT_INDEX32	DXGI_FORMAT_R32_UINT
D3DFMT_Q16W16V16U16	DXGI_FORMAT_R16G16B16A16_SNORM
D3DFMT_MULTI2_ARGB8	Not available
D3DFMT_R16F	DXGI_FORMAT_R16_FLOAT
D3DFMT_G16R16F	DXGI_FORMAT_R16G16_FLOAT
D3DFMT_A16B16G16R16F	DXGI_FORMAT_R16G16B16A16_FLOAT
D3DFMT_R32F	DXGI_FORMAT_R32_FLOAT
D3DFMT_G32R32F	DXGI_FORMAT_R32G32_FLOAT
D3DFMT_A32B32G32R32F	DXGI_FORMAT_R32G32B32A32_FLOAT
D3DFMT_CxV8U8	Not available
D3DDECLTYPE_FLOAT1	DXGI_FORMAT_R32_FLOAT
D3DDECLTYPE_FLOAT2	DXGI_FORMAT_R32G32_FLOAT
D3DDECLTYPE_FLOAT3	DXGI_FORMAT_R32G32B32_FLOAT

Direct3D 9 Texture/Vertex/Index Format	Equivalent Direct3D 10 Format
D3DDECLTYPE_FLOAT4	DXGI_FORMAT_R32G32B32A32_FLOAT
D3DDECLTYPED3DCOLOR	Not available
D3DDECLTYPE_UBYTE4	DXGI_FORMAT_R8G8B8A8_UINT [4]
D3DDECLTYPE_SHORT2	DXGI_FORMAT_R16G16_SINT
D3DDECLTYPE_SHORT4	DXGI_FORMAT_R16G16B16A16_SINT
D3DDECLTYPE_UBYTE4N	DXGI_FORMAT_R8G8B8A8_UNORM
D3DDECLTYPE_SHORT2N	DXGI_FORMAT_R16G16_SNORM
D3DDECLTYPE_SHORT4N	DXGI_FORMAT_R16G16B16A16_SNORM
D3DDECLTYPE_USHORT2N	DXGI_FORMAT_R16G16_UNORM
D3DDECLTYPE_USHORT4N	DXGI_FORMAT_R16G16B16A16_UNORM
D3DDECLTYPE_UDEC3	Not available
D3DDECLTYPE_DEC3N	Not available
D3DDECLTYPE_FLOAT16_2	DXGI_FORMAT_R16G16_FLOAT
D3DDECLTYPE_FLOAT16_4	DXGI_FORMAT_R16G16B16A16_FLOAT

1. Use .r swizzle in a shader to duplicate red to other components to get Direct3D 9 behavior.
2. In Direct3D 9, the data was scaled up by 255.0f which can be done in shader code instead.
3. DXT2 and DXT3 are the same from an API perspective; DXT4 and DXT5 are the same from an API perspective. The only difference was premultiplied alpha, which can be tracked by an application and does not need a separate format.
4. A shader gets UINT values, but if Direct3D 9 style integral floats (0.0f, 1.0f... 255.f) are needed, UINT can be converted to float32 in a shader.

Related topics

Resources (Direct3D 10)

Effects (Direct3D 10)

Article • 06/11/2021 • 2 minutes to read

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, geometry and pixel 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.

- Organizing State in an Effect
- Effect System Interfaces
- Specializing Interfaces
- Rendering an Effect

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.

Related topics

Programming Guide for Direct3D 10

Organizing State in an Effect (Direct3D 10)

Article • 01/06/2021 • 3 minutes to read

With Direct3D 10, effect state for certain pipeline stages is organized by the following structures:

Pipeline State	Structure
Input Assembler	D3D10_INPUT_ELEMENT_DESC
Rasterization	D3D10_RASTERIZER_DESC
Output Merger	D3D10_BLEND_DESC and D3D10_DEPTH_STENCIL_DESC

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, and shader resource state. 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
- 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 10). 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 pipeline uses three different kinds of shaders.

- Vertex shaders - Operate on vertex data. One vertex in yields one vertex 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).

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 ID3D10Effect 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 10).

Techniques and Passes

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.

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 ) ) );
    }
}
```

The syntax for effect shaders is more fully detailed in Effect Technique Syntax (Direct3D 10).

Related topics

Effects (Direct3D 10)

Effect System Interfaces (Direct3D 10)

Article • 01/06/2021 • 2 minutes to read

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
ID3D10Effect Interface	Collection of one or more techniques for rendering.
ID3D10Include Interface	An interface for adding custom behaviors when reading include files.
ID3D10EffectPass Interface	A collection of state assignments.
ID3D10EffectPool Interface	Create a memory location for variables to be shared between effects.
ID3D10EffectTechnique Interface	A collection of one or more passes.

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
ID3D10EffectBlendVariable Interface	Get and set blend state.

Reflection Interfaces	Description
ID3D10EffectDepthStencilVariable Interface	Get and set depth-stencil state.
ID3D10EffectRasterizerVariable Interface	Get and set rasterizer state.
ID3D10EffectSamplerVariable Interface	Get and set sampler state.

Setting Effect Resources

Use these interfaces to get and set resources.

Reflection Interfaces	Description
ID3D10EffectConstantBuffer Interface	Access data in a texture buffer or constant buffer.
ID3D10EffectDepthStencilViewVariable Interface	Access data in a depth-stencil resource.
ID3D10EffectRenderTargetViewVariable Interface	Access data in a render target.
ID3D10EffectShaderResourceVariable Interface	Access data in a shader resource.

Setting Other Effect Variables

Use these interfaces to get and set state by the variable type.

Reflection Interfaces	Description
ID3D10EffectMatrixVariable Interface	Get and set a matrix.
ID3D10EffectScalarVariable Interface	Get and set a scalar.
ID3D10EffectShaderVariable Interface	Get and set a shader variable.
ID3D10EffectStringVariable Interface	Get and set a string.
ID3D10EffectType Interface	Get a variable type.
ID3D10EffectVectorVariable Interface	Get and set a vector.

All reflection interfaces derive from **ID3D10EffectVariable Interface**.

Related topics

Effects

Programming Guide for Direct3D 10

Specializing Interfaces (Direct3D 10)

Article • 01/06/2021 • 2 minutes to read

ID3D10EffectVariable Interface has a number of methods for casting the interface into the particular type of interface you need. The methods are of the form As*Type* 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 (from SimpleSample10 Sample) that gets these variables:

```
ID3D10EffectVariable* g_pVariable;
ID3D10EffectMatrixVariable* g_pmWorld;
ID3D10EffectScalarVariable* g_pfTime;

g_pVariable = g_pEffect10->GetVariableByName("g_mWorld");
g_pmWorld = g_pVariable->AsMatrix();
g_pfTime = g_pEffect10->GetVariableByName("g_fTime");
g_pfTime = g_pVariable->AsScalar();
```

By specializing the interfaces, you could reduce the code to a single call.

```
g_pmWorld = (g_pEffect10->GetVariableByName("g_mWorld"))->AsMatrix();
g_pfTime = (g_pEffect10->GetVariableByName("g_fTime"))->AsScalar();
```

Interfaces that inherit from **ID3D10EffectVariable Interface** also have these methods, but they have been designed to return invalid objects; only calls from **ID3D10EffectVariable Interface** return valid objects. Applications can test the returned object to see if it is valid by calling **ID3D10EffectVariable::IsValid**.

Related topics

Effects

Rendering an Effect (Direct3D 10)

Article • 06/16/2021 • 2 minutes to read

An effect can be used to store information, or to render using a group of state. Each technique specifies vertex shaders, geometry shaders, pixel shaders, shader state, sampler and texture 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. 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. 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 affect by applying a technique. These topics are all discussed in further detail below.

- Compile an Effect (Direct3D 10)
- Create an Effect (Direct3D 10)
- Set Effect State (Direct3D 10)
- Apply a Technique (Direct3D 10)

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, or organizing the variables that need to be updated by frequency. 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.

Related topics

Effects (Direct3D 10)

Compile an Effect (Direct3D 10)

Article • 01/06/2021 • 3 minutes to read

Once an effect has been authored, the first step is to compile the code to check for syntax problems. This is done by calling one of the compile APIs (like D3DX10CompileEffectFromFile, D3DX10CompileEffectFromResource, D3DX10CompileEffectFromMemory). These API's invoke the effect compiler fxc.exe which is the compiler used to compile 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). By the way, the effect compiler/hlsl compiler (fxc.exe) is in the SDK in the utilities folder so that you can compile your shaders (or effects) offline if you choose. See the documentation for running the compiler from the command line.

- Includes
- Macros
- HLSL Shader Flags
- FX Flags
- Checking Errors
- Related topics

Here's an example of compiling an effect file (from the BasicHLSL10 sample).

```
WCHAR str[MAX_PATH];
DXUTFindDXSDKMediaFileCch( str, MAX_PATH, L"BasicHLSL10.fx" );

hr = D3DX10CompileEffectFromFile( str, NULL, NULL, "fx_4_0",
    D3D10_SHADER_ENABLE_STRICTNESS, 0, pd3dDevice, NULL, NULL,
    &l_pBlob_Effect, &l_pBlob_Errors, NULL );
```

Includes

One parameter is an include interface. Generate one of these if you want to include a customized behavior when reading an include file. This custom behavior is executed each time an effect (that uses the include pointer) is created or when an effect (that uses the include pointer) is compiled. To implement customized include behavior, derive a class from the Include interface. This provides your class two methods: Open and Close. Implement the custom behavior in the Open and Close methods.

Macros

Effect compilation can also take a pointer to macros that are defined elsewhere. For example, suppose you were 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.

```
D3D_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 with a **D3D_SHADER_MACRO** struct.

Lastly, modify the compile effect call to take a pointer to the macros.

```
D3DX10CreateEffectFromFile( str, Shader_Macros, NULL,
    D3D10_SHADER_ENABLE_STRICTNESS, 0, pd3dDevice, NULL, NULL,
    &g_pEffect10, NULL );
```

HLSL Shader Flags

Shader flags specify shader constraints to the HLSL compiler. These flags impact the code generated by the shader compiler including:

- Size considerations: optimize the code.
- Debug considerations: including debug information, preventing flow control.
- Hardware considerations: the compile target and whether or not a shader can run on legacy hardware.

In general, these flags can be logically combined, assuming you have not specified two conflicting characteristics. For a listing of the flags see Effect Constants (Direct3D 10).

FX Flags

These flags used when creating an effect to define either compilation behavior or runtime effect behavior. For a listing of the flags see Effect Constants (Direct3D 10).

Checking Errors

If during compilation, an error occurs, the API returns an interface which contains the errors returned from the effect compiler. This interface is called ID3D10Blob. 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.

In this example, an error was introduced into the BasicHLSL.fx effect by copying the first variable declaration twice.

```
//-----------------------------------------------------------------------
// Global variables
//-----------------------------------------------------------------------
float4 g_MaterialAmbientColor;      // Material's ambient color

// Declare the same variable twice
float4 g_MaterialAmbientColor;      // Material's ambient color
```

This error caused 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'"

Since the error is returned in a LPVOID pointer, cast it to a character string in the watch window.

Here is the code used to return the error from the failed compile.

```
// Read the D3DX effect file
WCHAR str[MAX_PATH];
ID3D10Blob* l_pBlob_Effect = NULL;
ID3D10Blob* l_pBlob_Errors = NULL;
hr = DXUTFindDXSDKMediaFileCch( str, MAX_PATH, L"BasicHLSL10.fx" );
hr = D3DX10CompileEffectFromFile( str, NULL, NULL,
    D3D10_SHADER_ENABLE_STRICTNESS, 0, NULL,
```

```
    &l_pBlob_Effect, &l_pBlob_Errors );

LPVOID l_pError = NULL;
if( l_pBlob_Errors )
{
    l_pError = l_pBlob_Errors->GetBufferPointer();
    // then cast to a char* to see it in the locals window
}
```

Related topics

Rendering an Effect (Direct3D 10)

Create an Effect (Direct3D 10)

Article • 01/06/2021 • 2 minutes to read

An effect is created by loading it into the effects framework. If the effect has never been compiled, it will be compiled when it is created. Effects that are already loaded into memory can be created by calling **D3DX10CreateEffectFromMemory**. The following code example uses **D3DX10CreateEffectFromFile** to create an effect from a file.

```
ID3D10Effect* g_pEffect10 = NULL;

// Read the effect file
D3DX10CreateEffectFromFile( "BasicHLSL10.fx", NULL, NULL,
  D3D10_SHADER_ENABLE_STRICTNESS, 0, pd3dDevice, NULL, NULL,
  &g_pEffect10, NULL );
```

Reading an effect requires the same parameters as compiling an effect, plus a device and a pool.

Related topics

Rendering an Effect (Direct3D 10)

Set Effect State (Direct3D 10)

Article • 06/18/2021 • 3 minutes to read

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();


OnD3D10CreateDevice()
{
  ...
    g_pLightDir = g_pEffect10->GetVariableByName( "g_LightDir" )-
>AsVector();
    g_pLightDiffuse = g_pEffect10->GetVariableByName( "g_LightDiffuse" )-
>AsVector();
    g_pmWorldViewProjection = g_pEffect10->GetVariableByName(
        "g_mWorldViewProjection" )->AsMatrix();
    g_pmWorld = g_pEffect10->GetVariableByName( "g_mWorld" )->AsMatrix();
    g_pfTime = g_pEffect10->GetVariableByName( "g_fTime" )->AsScalar();
    g_pMaterialAmbientColor = g_pEffect10-
>GetVariableByName("g_MaterialAmbientColor")->AsVector();
    g_pMaterialDiffuseColor = g_pEffect10->GetVariableByName(
        "g_MaterialDiffuseColor" )->AsVector();
    g_pnNumLights = g_pEffect10->GetVariableByName( "g_nNumLights" )-
>AsScalar();
}
```

Third, use the update methods to set the value of the variables in the application in the effect variables.

```
OnD3D10FrameRender()
{
    ...
    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 **ID3D10EffectVariable Interface** that has been cast as a sampler interface.

```
D3D10_SAMPLER_DESC sampler_desc;
ID3D10EffectVariable* l_pD3D10EffectVariable = NULL;
if( g_pEffect10 )
{
    l_pD3D10EffectVariable = g_pEffect10->GetVariableByName(
"MeshTextureSampler" );
    if( l_pD3D10EffectVariable )
        hr = (l_pD3D10EffectVariable->AsSampler())->GetBackingStore( 0,
            &sampler_desc );
}
```

The other way is to get the sampler state from an **ID3D10SamplerState Interface**.

```
ID3D10SamplerState* l_ppSamplerState = NULL;
D3D10_SAMPLER_DESC sampler_desc;
ID3D10EffectVariable* l_pD3D10EffectVariable = NULL;
if( g_pEffect10 )
{
    l_pD3D10EffectVariable = g_pEffect10->GetVariableByName(
"MeshTextureSampler" );
    if( l_pD3D10EffectVariable )
    {
        hr = (l_pD3D10EffectVariable->AsSampler())->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.

```
technique10 RenderSceneWithTexture1Light
{
    pass P0
    {
        SetVertexShader( CompileShader( vs_4_0, RenderSceneVS( 1, true, true
) ) );
        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 ID3D10Device::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 SetXXXShader calls compiles their associated shader function and returns a pointer to the compiled shader.

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).

```
ID3D10EffectShaderResourceVariable* g_ptxDiffuse = NULL;

    // Obtain variables
    g_ptxDiffuse = g_pEffect10->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.

```
OnD3D10FrameRender()
{
  ID3D10ShaderResourceView* pDiffuseRV = NULL;

   ...
   pDiffuseRV = g_Mesh10.GetMaterial(pSubset->MaterialID)->pDiffuseRV10;
   g_ptxDiffuse->SetResource( pDiffuseRV );
   ...
}
```

For more information about viewing resources, see Texture Views (Direct3D 10).

Related topics

Rendering an Effect (Direct3D 10)

Apply a Technique (Direct3D 10)

Article • 01/06/2021 • 2 minutes to read

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 };
ID3D10RenderTargetView* pRTV = DXUTGetD3D10RenderTargetView();
pd3dDevice->ClearRenderTargetView( pRTV, ClearColor );
ID3D10DepthStencilView* pDSV = DXUTGetD3D10DepthStencilView();
pd3dDevice->ClearDepthStencilView( pDSV, D3D10_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).

```
D3D10_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 10)

Performance Considerations (Direct3D 10)

Article • 01/06/2021 • 2 minutes to read

Using Effect Pools

It is common for rendering pipelines to use many shaders to render different object types and special effects. The shader are a mixture of states that a common among all the shaders such as a world matrix or a light position, and other state that is specific to each shader such as an object's diffuse color, or specular highlight calculation. An effect pool is a place in memory to store state that is used across many shaders, as well as common device objects such as shaders, render state objects and constant buffers. The performance improvement results from updating the common state once for all the shaders that need that state.

An effect pool is a shared memory location for effect state. A pool is created similarly to an effect; it can be created from memory (or a file or a resource). This leads to two different types of effects: a global effect that does not depend on state in other effect versus a child effect which depends on state in another effect.

You specify whether an effect is a global effect (the default case) or a child effect (by supplying the D3D10_EFFECT_COMPILE_CHILD_EFFECT flag) when the effect is created. A global effect can serve as an effect pool; a child effect cannot be an effect pool.

Related topics

Rendering an Effect

Effects (Direct3D 10)

Direct3D 9 to Direct3D 10 Considerations (Direct3D 10)

Article • 01/06/2021 • 23 minutes to read

The following page provides a basic outline of key differences between Direct3D 9 and Direct3D 10. The outline below provides some insight to assist developers with Direct3D 9 experience to explore and relate to Direct3D 10.

Although the info in this topic compares Direct3D 9 with Direct3D 10, because Direct3D 11 builds on the improvements made in Direct3D 10 and 10.1, you also need this info to migrate from Direct3D 9 to Direct3D 11. For info about moving beyond Direct3D 10 to Direct3D 11, see Migrating to Direct3D 11.

- Overview of the Major Structural Changes in Direct3D 10
  - Removal of Fixed Function
  - Device Object Creation Time Validation
- Engine Abstractions / Separation
  - Direct Removal of Direct3D 9 Dependencies
- Tricks for Quickly Resolving Application Build Issues
  - Overriding Direct3D 9 Types
  - Resolving Link Issues
  - Simulating Device CAPs
- Driving the Direct3D 10 API
  - Resource Creation
  - Views
  - Static versus Dynamic Resource Access
  - Direct3D 10 Effects
  - HLSL without Effects
  - Shader Compilation
  - Creation of Shader Resources
  - Shader Reflection Layer Interface
  - Input Assembler Layouts - Vertex Shader / Input Stream Linkage
  - Impact of Shader Dead Code Removal
  - Vertex Shader Input Structure Example
  - State Object Creation
- Porting Textures
  - File Formats Supported
  - Mapping Texture Formats
- Porting Shaders
  - Direct3D 10 Shaders are Authored in HLSL

    - Shader Signatures and Linkage
    - HLSL Shader Stage linkages
    - Constant Buffers
    - User clip planes in HLSL on feature level 9 and higher
- Additional Direct3D 10 Differences to Watch For
    - Integers as Input
    - Mouse Cursors
    - Mapping Texels to Pixels in Direct3D 10
    - Reference Counting Behavior Changes
    - Test Cooperative Level
    - StretchRect
- Additional Direct3D 10.1 Differences
- Related topics

Overview of the Major Structural Changes in Direct3D 10

- Removal of Fixed Function
- Device Object Creation Time Validation

The process of rendering using the Direct3D 10 device is structurally similar to Direct3D 9.

- Set a vertex stream source
- Set input layout in Direct3D 10 (set vertex stream declaration in Direct3D 9)
- Declare primitive topology
- Set textures
- Set state objects
- Set shaders
- Draw

The **Draw** call ties the operations together; the ordering of calls prior to the Draw call is arbitrary. The major differences in the Direct3D 10 API design are as follows:

- Removal of Fixed Function
- Removal of CAPS bits - Direct3D 10's base feature set is guaranteed
- Stricter management of: resource access, device state, shader constants, shader linkage (inputs and outputs to shaders) between stages
- API entry point name changes reflect the use of virtual GPU memory (Map() instead of Lock()).
- A debug layer can be added to the device at creation time

- The primitive topology is now an explicit state (separated from the **Draw** call)
- Explicit shader constants are now stored in constant buffers
- Shader authoring is done entirely in HLSL. The HLSL compiler now resides in the primary Direct3D 10 DLL.
- New programmable stage - the geometry shader
- Removal of BeginScene()/EndScene()
- Common 2D, focus and adapter-management functionality implemented in a new component: DXGI

Removal of Fixed Function

It is sometimes surprising that even in a Direct3D 9 engine that fully exploits the programmable pipeline, there remains a number of areas that depend on the fixed-function (FF) pipeline. The most common areas are usually related to screen-space aligned rendering for UI. It is for this reason that you are likely to need to build a FF emulation shader or set of shaders which provide the necessary replacement behaviors.

This documentation contains a white paper containing replacement shader sources for the most common FF behaviors (see Fixed Function EMU Sample). Some fixed-function pixel behavior including alpha test has been moved into shaders.

Device Object Creation Time Validation

The Direct3D 10 pipeline has been redesigned from the ground up in hardware and software with a primary intention to reduce CPU overhead (at Draw time). To reduce costs, all types of device data have been assigned an object with explicit creation methods provided by the device itself. This enables strict data validation at object creation time instead of during the Draw call as it often does with Direct3D 9.

Engine Abstractions / Separation

Applications, including Games, that wish to support both Direct3D 9 and Direct3D 10 need to have the rendering layers abstracted from the rest of the code base. There are many ways to achieve this but key to all of them is the design of the abstraction layer for the lower-level Direct3D device. All systems should communicate to the hardware through the common layer which is designed to provide GPU resource and low-level type management.

Direct Removal of Direct3D 9 Dependencies

When porting large, previously tested code bases, it is important to minimize the amount of code changes to those which are absolutely necessary to preserve previously tested behaviors in the code. Best practices include clearly documenting where items change using comments. It's often useful to have a commenting standard for this work which enables quick navigation through the code base.

The following is an example list of standard single line / start block comments which could be used for this work.

Item	Description
// Direct3D 10 REMOVED	Use this where lines / blocks of code are removed
// Direct3D 10 NEEDS UPDATE	It helps to added additional notes to the NEED UPDATE comment that suggests what work / new API should be used for later visits to the code for behavior conversion. Heavy use of assert(**false**) should also be used where \\ Direct3D 10 NEEDS UPDATE occurs to ensure you do not unknowingly run errant code
// Direct3D 10 CHANGED	Areas where major changes have occurred should be kept for future reference but commented out
// Direct3D 10 END	End code block qualifier

For multiple lines of source you should use the C style /* */ comments too but add the relevant start / end comments around these areas.

Tricks for Quickly Resolving Application Build Issues

- Overriding Direct3D 9 Types
- Resolving Link Issues
- Simulating Device CAPs

Overriding Direct3D 9 Types

It may be useful to insert a high level header file containing type definitions / overrides for Direct3D 9 base types which are no longer supported by the Direct3D 10 headers. This will help you minimize the amount of changes in the code and interfaces where there is a direct mapping from a Direct3D 9 type to the newly defined Direct3D 10 type. This approach is also useful for keeping code behaviors together in one source file. In this case, it is a good idea to define version neutral / generally named types which describe common constructs used for rendering yet span both Direct3D 9 and Direct3D 10 APIs. For example:

```
#if defined(D3D9)
typedef IDirect3DIndexBuffer9   IDirect3DIndexBuffer;
typedef IDirect3DVertexBuffer9  IDirect3DVertexBuffer;
#else //D3D10
typedef ID3D10Buffer            IDirect3DIndexBuffer;
typedef ID3D10Buffer            IDirect3DVertexBuffer
#endif
```

Other Direct3D 10 specific examples of this include:

```
typedef ID3D10TextureCube   IDirect3DCubeTexture;
typedef ID3D10Texture3D     IDirect3DVolumeTexture;
typedef D3D10_VIEWPORT      D3DVIEWPORT;
typedef ID3D10VertexShader  IDirect3DVertexShader;
typedef ID3D10PixelShader   IDirect3DPixelShader;
```

Resolving Link Issues

It is advisable to develop Direct3D 10 and Windows Vista applications using the latest version of Microsoft Visual Studio. However, it is possible to build a Windows Vista application which depends on Direct3D 10 using the earlier 2003 version of Visual Studio. Direct3D 10 is a Windows Vista platform component which has dependencies (as with the Server 2003 SP1 platform SDK) on the following lib: BufferOverflowU.lib is needed to solve any buffer_security check linker issues.

Simulating Device CAPs

Many applications contain areas of code which depend on device CAPS data being available. Work around this by overriding device enumeration and forcing device CAPS

to sensible values. Plan to re-visit areas where there are dependencies on CAPS later for full removal of CAPS where possible.

Driving the Direct3D 10 API

This section focuses on the behavior changes caused by the Direct3D 10 API.

- Resource Creation
- Views
- Static versus Dynamic Resource Access
- Direct3D 10 Effects
- HLSL without Effects
- Shader Compilation
- Creation of Shader Resources
- Shader Reflection Layer Interface
- Input Assembler Layouts - Vertex Shader / Input Stream Linkage
- Impact of Shader Dead Code Removal
- Vertex Shader Input Structure Example
- State Object Creation

Resource Creation

The Direct3D 10 API has designed resources as generic buffer types which have specific bind flags according to the planned usage. This design point was chosen to facilitate near ubiquitous access of resources in the Pipeline for scenarios such as rendering to a vertex buffer then instantly drawing the results without interrupting the CPU. The following example demonstrates allocation of vertex buffers and index buffer where you can see that the resource description only differs by the GPU resource bind flags.

The Direct3D 10 API has provided texture helper methods for explicitly creating texture type resources but as you can imagine, these are really helper functions.

- CreateTexture2D()
- CreateTextureCube()
- CreateTexture3D()

When targeting Direct3D 10, you are likely to want to allocate more items during resource creation time than you are used to with Direct3D 9. This will become most apparent with the creation of Render Target buffers and Textures where you need to also create a view for accessing the buffer and setting the resource on the device.

Tutorial 1: Direct3D 10 Basics

> **Note**
>
> Direct3D 10 and later versions of Direct3D extend the DDS file format to support new DXGI formats, texture arrays, and cube-map arrays. For more information about the DDS file format extension, see the **Programming Guide for DDS**.

Views

A view is a specifically typed, interface to data stored in a pixel buffer. A resource can have several views allocated on it at once and this feature is highlighted in the Single Pass Render to Cubemap sample contained in this SDK.

Programmers Guide page on Resource Access

CubeMap Sample

Static versus Dynamic Resource Access

For best performance, applications should partition their data use in terms of the static vs dynamic nature of the data. Direct3D 10 has been designed to take advantage of this approach and as such, the access rules for resources have been significantly tightened up over Direct3D 9. For static resources you should ideally populate the resource with its data during creation time. If your engine has been designed around the Create, Lock, Fill, Unlock design point of Direct3D 9, you may defer population from Create time by using a staging resource and the UpdateSubResource method on the resource interface.

Direct3D 10 Effects

Use of the Direct3D 10 Effects system is outside the scope of this article. The system has been written to take full advantage of the architectural benefits that Direct3D 10 provides. See the Effects (Direct3D 10) section for more detail on its use.

HLSL without Effects

The Direct3D 10 Shader pipeline may be driven without the use of the Direct3D 10 Effects system. Note that in this instance, all constant buffer, shader, sampler and texture binding must be managed by the application itself. See the sample link and following sections of this document for more detail:

HLSL without Effects sample

Shader Compilation

The Direct3D 10 HLSL compiler brings enhancements to the HLSL language definition and therefore has the ability to operate in 2 modes. For complete support of Direct3D 9 style intrinsic functions and semantics, compilation should be invoked using the COMPATIBILITY MODE flag which can be specified on a per compile basis.

The shader model 4.0 specific HLSL language semantics and intrinsic functions for Direct3D 10 can be found at HLSL. Major changes in syntax from Direct3D 9 HLSL to take the most notice of are in the area of texture access. The new syntax is the only form supported by the compiler outside of compatibility mode.

> **Note**
>
> The Direct3D 10 compiler-type APIs (**D3D10CompileShader** and **D3D10CompileEffectFromMemory**) are supplied by the Direct3D 10, 10.1, and 11 runtimes that run in Windows Vista and later. The Direct3D 10 compiler-type APIs have the same functionality as the HLSL compiler that is shipped in the DirectX SDK (December 2006). This HLSL compiler does not support the Direct3D 10.1 profiles (vs_4_1, ps_4_1, gs_4_1, fx_4_1), and is missing a number of optimizations and improvements. You can get an HLSL compiler that supports the Direct3D 10.1 profiles from the latest legacy **DirectX SDK release**. For info about the legacy DirectX SDK, see **Where is the DirectX SDK?**. You can get the latest HLSL Fxc.exe command-line compiler and **D3DCompiler** APIs from the Windows SDK.

Creation of Shader Resources

The creation of compiled shader instances outside of the Direct3D 10 Effects system is done in a very similar manner to Direct3D 9 however, in Direct3D 10, it is important to keep the Shader Input signature around for later use. The signature is returned by default as part of the shader blob but may be extracted to reduce the memory requirements if needed. For more details, see Using Shaders in Direct3D 10.

Shader Reflection Layer Interface

The shader reflection layer is the interface by which information about the shader requirements may be obtained. This is particularly useful when creating Input Assembly linkages (see below) where you may need to traverse the shader input requirements to ensure you are supplying the correct input structure to the shader. You can create an instance of the reflection layer interface at the same time as creating an instance of a compiled shader.

The shader reflection layer replaces D3DX9 methods that provide similar functionality. For example **IsParameterUsed** is replaced by the **GetDesc** method.

Input Assembler Layouts - Vertex Shader / Input Stream Linkage

The Input Assembler (IA) replaces the Direct3D 9 style Vertex Stream Declaration and it's description structure is very similar in form. The main difference that the IA brings is that the IA layout object created must directly map to a specific format of shader input signature. The mapping object created to link the input stream to shader may be used across any number of shaders where the shader input signature matches that of the shader used to create the Input Layout.

To best drive the pipeline with static data, you should consider the permutations of input stream format to possible shader input signatures and create the IA layout object instances as early as is possible and reuse them where possible.

Impact of Shader Dead Code Removal

The following section details a significant difference between Direct3D 9 and Direct3D 10 that is likely to require careful handling in your engine code. Shaders which contain conditional expressions often have certain code paths removed as part of the compilation process. In Direct3D 9, two kinds of inputs may be removed (marked for removal) when unused: signature inputs (like the example below) and constant inputs. If the end of the constant buffer contains unused entries, the size declaration in the shader will reflect the size of the constant buffer without the unused entries at the end. Both of these kinds of inputs remain in signatures or constant buffers Direct3D 10 with a special exception in the case of unused constant inputs at the end of a constant buffer. This may have an impact on the engine when handling large shaders and creating input layouts. Elements which are removed by dead code optimizations in the compiler must still be declared in the input structure. The following example demonstrates this:

Vertex Shader Input Structure Example

```
struct VS_INPUT
{
float4 pos: SV_Position;
float2 uv1 : Texcoord1;
float2 uv2 : Texcoord2; *
};
```

* Direct3D 9 dead code removal would remove the declaration in the shader due to conditional dead code removal

```
float4x4  g_WorldViewProjMtx;
static const bool g_bLightMapped = false; // define a compile time constant

VS_INPUT main(VS_INPUT i)
{
    VS_INPUT o;
    o.pos = mul( i.pos, g_WorldViewProjMtx);
    o.uv1 = i.uv1;
    if ( g_bLightMapped )
    {
        o.uv2 = i.uv2;
    }
    return o;
}
```

Or, you could make it even more obvious that the constant is a compile time constant with the following declaration:

```
#define LIGHT_MAPPED false
```

In the above example, under Direct3D 9, the uv2 element would be removed due to dead code optimizations in the compiler. Under Direct3D 10, the dead code will still be removed but the shader input assembler layout requires the definition of the input data to exist. The shader reflection layer provides the means to handle this situation in a generic manner whereby you can traverse the shader input requirements and ensure that you provide a full description of the Input Stream to shader signature mapping.

Here's an example function to detect the existence of a semantic name/index in a function signature:

```
// Returns true if the SemanticName / SemanticIndex is used in the input
signature.
// pReflector is a previously acquired shader reflection interface.
bool IsSignatureElementExpected(ID3D10ShaderReflection *pReflector, const
LPCSTR SemanticName, UINT SemanticIndex)
{
    D3D10_SHADER_DESC               shaderDesc;
    D3D10_SIGNATURE_PARAMETER_DESC  paramDesc;

    Assert(pReflector);
    Assert(SemanticName);

    pReflector->GetDesc(&shaderDesc);

    for (UINT k=0; k<shaderDesc.InputParameters; k++)
    {
        pReflector->GetInputParameterDesc( k, &paramDesc);
        if (wcscmp( SemanticName, paramDesc.SemanticName)==0 &&
paramDesc.SemanticIndex == SemanticIndex)
            return true;
    }

    return false;
}
```

State Object Creation

When porting engine code, it may help to initially use a default set of state objects and disable all Direct3D 9 device render state / texture state setting. This will cause rendering artifacts but is the quickest way to get things up and running. You can later construct a state object management system which can use a compound key to enable maximum reuse of the number of state objects being used.

Porting Textures

File Formats Supported

The **D3DXxxCreateXXX** and **D3DXxxSaveXXX** functions that create or save a texture from or to a graphics file (for example, D3DX10CreateTextureFromFile) support a different set of file formats in Direct3D 10 than they did in Direct3D 9:

File Format	Direct3D 9	Direct3D 10
.bmp	x	x
.jpg	x	x

File Format	Direct3D 9	Direct3D 10
.tga	x	
.png	x	x
.dds	x	x
.ppm	x	
.dib	x	
.hdr	x	
.pfm	x	
.tiff		x
.gif		x
.tif		x

For details, compare the Direct3D 9 D3DXIMAGE_FILEFORMAT enumerations with the D3DX10_IMAGE_FILE_FORMAT enumerations for Direct3D 10.

> **Note**
>
> The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8. For texture file processing, we recommend that you use DirectXTex.

Mapping Texture Formats

The following table shows the mapping of texture formats from Direct3D 9 to Direct3D 10. Any content in formats not available in DXGI will need to be converted by utility routines.

Direct3D 9 Format	Direct3D 10 Format
D3DFMT_UNKNOWN	DXGI_FORMAT_UNKNOWN
D3DFMT_R8G8B8	Not available

Direct3D 9 Format	Direct3D 10 Format
D3DFMT_A8R8G8B8	DXGI_FORMAT_B8G8R8A8_UNORM & DXGI_FORMAT_B8G8R8A8_UNORM_SRGB[1]
D3DFMT_X8R8G8B8	DXGI_FORMAT_B8G8R8X8_UNORM & DXGI_FORMAT_B8G8R8X8_UNORM_SRGB[1]
D3DFMT_R5G6B5	DXGI_FORMAT_B5G6R5_UNORM[2]
D3DFMT_X1R5G5B5	Not available
D3DFMT_A1R5G5B5	DXGI_FORMAT_B5G5R5A1_UNORM[2]
D3DFMT_A4R4G4B4	DXGI_FORMAT_B4G4R4A4_UNORM[3]
D3DFMT_R3G3B2	Not available
D3DFMT_A8	DXGI_FORMAT_A8_UNORM
D3DFMT_A8R3G3B2	Not available
D3DFMT_X4R4G4B4	Not available
D3DFMT_A2B10G10R10	DXGI_FORMAT_R10G10B10A2
D3DFMT_A8B8G8R8	DXGI_FORMAT_R8G8B8A8_UNORM & DXGI_FORMAT_R8G8B8A8_UNORM_SRGB
D3DFMT_X8B8G8R8	Not available
D3DFMT_G16R16	DXGI_FORMAT_R16G16_UNORM
D3DFMT_A2R10G10B10	Not available
D3DFMT_A16B16G16R16	DXGI_FORMAT_R16G16B16A16_UNORM
D3DFMT_A8P8	Not available
D3DFMT_P8	Not available
D3DFMT_L8	DXGI_FORMAT_R8_UNORM Note: Use .r swizzle in shader to duplicate red to other components to get D3D9 behavior.
D3DFMT_A8L8	DXGI_FORMAT_R8G8_UNORM Note: Use swizzle .rrrg in shader to duplicate red and move green to the alpha components to get D3D9 behavior.
D3DFMT_A4L4	Not available
D3DFMT_V8U8	DXGI_FORMAT_R8G8_SNORM
D3DFMT_L6V5U5	Not available

Direct3D 9 Format	Direct3D 10 Format
D3DFMT_X8L8V8U8	Not available
D3DFMT_Q8W8V8U8	DXGI_FORMAT_R8G8B8A8_SNORM
D3DFMT_V16U16	DXGI_FORMAT_R16G16_SNORM
D3DFMT_W11V11U10	Not available
D3DFMT_A2W10V10U10	Not available
D3DFMT_UYVY	Not available
D3DFMT_R8G8_B8G8	DXGI_FORMAT_G8R8_G8B8_UNORM (in DX9 the data was scaled up by 255.0f, but this can be handled in shader code).
D3DFMT_YUY2	Not available
D3DFMT_G8R8_G8B8	DXGI_FORMAT_R8G8_B8G8_UNORM (in DX9 the data was scaled up by 255.0f, but this can be handled in shader code).
D3DFMT_DXT1	DXGI_FORMAT_BC1_UNORM & DXGI_FORMAT_BC1_UNORM_SRGB
D3DFMT_DXT2	DXGI_FORMAT_BC1_UNORM & DXGI_FORMAT_BC1_UNORM_SRGB Note: DXT1 and DXT2 are the same from an API/hardware perspective... only difference was 'premultiplied alpha', which can be tracked by an application and doesn't need a separate format.
D3DFMT_DXT3	DXGI_FORMAT_BC2_UNORM & DXGI_FORMAT_BC2_UNORM_SRGB
D3DFMT_DXT4	DXGI_FORMAT_BC2_UNORM & DXGI_FORMAT_BC2_UNORM_SRGB Note: DXT3 and DXT4 are the same from an API/hardware perspective... only difference was 'premultiplied alpha', which can be tracked by an application and doesn't need a separate format.
D3DFMT_DXT5	DXGI_FORMAT_BC3_UNORM & DXGI_FORMAT_BC3_UNORM_SRGB
D3DFMT_D16 & D3DFMT_D16_LOCKABLE	DXGI_FORMAT_D16_UNORM
D3DFMT_D32	Not available
D3DFMT_D15S1	Not available
D3DFMT_D24S8	Not available
D3DFMT_D24X8	Not available
D3DFMT_D24X4S4	Not available
D3DFMT_D16	DXGI_FORMAT_D16_UNORM

Direct3D 9 Format	Direct3D 10 Format
D3DFMT_D32F_LOCKABLE	DXGI_FORMAT_D32_FLOAT
D3DFMT_D24FS8	Not available
D3DFMT_S1D15	Not available
D3DFMT_S8D24	DXGI_FORMAT_D24_UNORM_S8_UINT
D3DFMT_X8D24	Not available
D3DFMT_X4S4D24	Not available
D3DFMT_L16	DXGI_FORMAT_R16_UNORM Note: Use .r swizzle in shader to duplicate red to other components to get D3D9 behavior.
D3DFMT_INDEX16	DXGI_FORMAT_R16_UINT
D3DFMT_INDEX32	DXGI_FORMAT_R32_UINT
D3DFMT_Q16W16V16U16	DXGI_FORMAT_R16G16B16A16_SNORM
D3DFMT_MULTI2_ARGB8	Not available
D3DFMT_R16F	DXGI_FORMAT_R16_FLOAT
D3DFMT_G16R16F	DXGI_FORMAT_R16G16_FLOAT
D3DFMT_A16B16G16R16F	DXGI_FORMAT_R16G16B16A16_FLOAT
D3DFMT_R32F	DXGI_FORMAT_R32_FLOAT
D3DFMT_G32R32F	DXGI_FORMAT_R32G32_FLOAT
D3DFMT_A32B32G32R32F	DXGI_FORMAT_R32G32B32A32_FLOAT
D3DFMT_CxV8U8	Not available
D3DDECLTYPE_FLOAT1	DXGI_FORMAT_R32_FLOAT
D3DDECLTYPE_FLOAT2	DXGI_FORMAT_R32G32_FLOAT
D3DDECLTYPE_FLOAT3	DXGI_FORMAT_R32G32B32_FLOAT
D3DDECLTYPE_FLOAT4	DXGI_FORMAT_R32G32B32A32_FLOAT
D3DDECLTYPED3DCOLOR	Not available
D3DDECLTYPE_UBYTE4	DXGI_FORMAT_R8G8B8A8_UINT Note: Shader gets UINT values, but if Direct3D 9 style integral floats are needed (0.0f, 1.0f... 255.f), UINT can just be converted to float32 in shader.

Direct3D 9 Format	Direct3D 10 Format
D3DDECLTYPE_SHORT2	DXGI_FORMAT_R16G16_SINT Note: Shader gets SINT values, but if Direct3D 9 style integral floats are needed, SINT can just be converted to float32 in shader.
D3DDECLTYPE_SHORT4	DXGI_FORMAT_R16G16B16A16_SINT Note: Shader gets SINT values, but if Direct3D 9 style integral floats are needed, SINT can just be converted to float32 in shader.
D3DDECLTYPE_UBYTE4N	DXGI_FORMAT_R8G8B8A8_UNORM
D3DDECLTYPE_SHORT2N	DXGI_FORMAT_R16G16_SNORM
D3DDECLTYPE_SHORT4N	DXGI_FORMAT_R16G16B16A16_SNORM
D3DDECLTYPE_USHORT2N	DXGI_FORMAT_R16G16_UNORM
D3DDECLTYPE_USHORT4N	DXGI_FORMAT_R16G16B16A16_UNORM
D3DDECLTYPE_UDEC3	Not available
D3DDECLTYPE_DEC3N	Not available
D3DDECLTYPE_FLOAT16_2	DXGI_FORMAT_R16G16_FLOAT
D3DDECLTYPE_FLOAT16_4	DXGI_FORMAT_R16G16B16A16_FLOAT
FourCC 'ATI1'	DXGI_FORMAT_BC4_UNORM
FourCC 'ATI2'	DXGI_FORMAT_BC5_UNORM

[1]DXGI 1.1, which is included in the Direct3D 11 runtime, includes BGRA formats. However, support for these formats is optional for Direct3D 10 and 10.1 devices with drivers that are implemented to the Windows Display Driver Model (WDDM) for Windows Vista (WDDM 1.0). Consider using DXGI_FORMAT_R8G8B8A8_UNORM instead. Alternatively, you can create your device with **D3D10_CREATE_DEVICE_BGRA_SUPPORT** to ensure that you only support computers with the Direct3D 11.0 runtime and a WDDM 1.1 driver or higher installed.

[2]DXGI 1.0 defined 5:6:5 and 5:5:5:1 formats, but they were not supported by the Direct3D 10.x or Direct3D 11.0 runtime. These formats are optionally supported with DXGI 1.2 in the DirectX 11.1 runtime, which is required for feature level 11.1 video adapters and WDDM 1.2 (display driver model starting with Windows 8) drivers and already supported on 10level9 feature levels.

[3]DXGI 1.2 introduced the 4:4:4:4 format. This format is optionally supported in the DirectX 11.1 runtime, which is required for feature level 11.1 video adapters and WDDM 1.2 drivers and already supported on 10level9 feature levels.

For uncompressed formats, DXGI has limited the support for arbitrary pixel format patterns; all uncompressed formats must be of type RGBA. This might require swizzling of existing assets pixel formats, which we recommend that you compute as an offline pre-process pass where possible.

Porting Shaders

Direct3D 10 Shaders are Authored in HLSL

Direct3D 10 limits the use of assembly language to that of debugging purposes only, therefore any hand written assembly shaders used in Direct3D 9 will need to be converted to HLSL.

Shader Signatures and Linkage

We discussed the requirements for Input Assembly linkage to Vertex shader input signatures earlier in this document (see above). Note that the Direct3D 10 runtime has also tightened the requirements for stage to stage linkage between shaders. This change will affect shader sources where the binding between stages may not have been fully described under Direct3D 9. For example:

```
VS_OUTPUT                        PS_INPUT
float4   pos : SV_POSITION;      float4 pos : SV_POSITION;
float4   uv1 : TEXCOORD1;        float4 uv1 : TEXCOORD1;
float4x3 tangentSp : TEXCOORD2;  float4 tangent : TEXCOORD2; *
float4   Color : TEXCOORD6;      float4 color : TEXCOORD6;
```

* Broken VS - PS Linkage - even though the pixel shader may not be interested in the full matrix, the linkage must specify the full float4x3.

Note, the linkage semantics between stages must match exactly however, the target stages inputs may be a prefix of the values being output. In the example above, the pixel shader could have position and texcoord1 as the only inputs, but it could not have the position and texcoord2 as the only inputs due to the ordering constraints.

HLSL Shader Stage linkages

Linkage between shaders may occur at any of the following points in the pipeline:

- Input Assembler to Vertex Shader
- Vertex Shader to Pixel Shader
- Vertex Shader to Geometry Shader
- Vertex Shader to Stream Output
- Geometry Shader to Pixel Shader
- Geometry Shader to Stream Out

Constant Buffers

For ease of porting content from Direct3D 9 an initial approach to constant management outside of the Effects system might involve the creation of a single constant buffer containing all the required constants. It is important for performance to order constants into buffers by the expected frequency of update. This organization will reduce the amount of redundant constant sets to a minimum.

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.

Additional Direct3D 10 Differences to Watch For

Integers as Input

In Direct3D 9, there was no real hardware support for integer data types, however Direct3D 10 hardware supports explicit integer types. If you have floating-point data in your vertex buffer, then you must have a float input. Otherwise an integer type will be the bit pattern representation of the floating-point value. An integer type is not allowed for a pixel shader input unless the value is marked for no interpolation (see Interpolation Modifiers).

Mouse Cursors

On previous versions of Windows, the standard GDI mouse cursor routines did not operate correctly on all full-screen exclusive devices. The **SetCursorProperties**, **ShowCursor**, and **SetCursorPosition** APIs were added to handle these cases. Since Windows Vista's version of GDI fully understands DXGI surfaces, there is no need for this specialized mouse cursor API so there is no Direct3D 10 equivalent. Direct3D 10 applications should instead use the standard GDI mouse cursor routines for mouse cursors.

Mapping Texels to Pixels in Direct3D 10

In Direct3D 9, texel centers and pixel centers were a half unit apart (see Directly Mapping Texels to Pixels (Direct3D 9)). In Direct3D 10, texel centers are already at half-units, so there is no need to shift vertex coordinates at all.

Rendering full-screen quads is more straight-forward with Direct3D 10. Full-screen quads should be defined in clipspace (-1,1) and simply passed through the vertex shader with no changes. This way, there is no need to reload your vertex buffer every time the screen resolution changes, and there's no extra work in the pixel shader to manipulate the texture coordinates.

Reference Counting Behavior Changes

Unlike previous Direct3D versions, the various Set functions will not hold a reference to the devices objects. This means that the application must ensure that it holds a reference on the object for as long as it would like that object to be bound to the pipeline. When the ref count of the object drops to zero, then the object will be unbound from the pipeline as it is destroyed. This style of reference holding is also known as weak-reference holding, so therefore each bind location on the Device object holds a weak-reference to the interface/ object. Unless explicitly mentioned otherwise, this behavior should be assumed for all Set methods. Whenever destruction of an object causes a bind point to be set to **NULL** out, the Debug Layer will issue a warning. Note, calls to device Get methods such as **OMGetRenderTargets** will increase the reference count of objects being returned.

Test Cooperative Level

The functionality of the Direct3D 9 API **TestCooperativeLevel** is analogous to setting the DXGI_PRESENT_TEST when calling **Present**.

StretchRect

A function similar to the Direct3D 9 IDirect3DDevice9::StretchRect method is not available in Direct3D 10 and 10.1. To copy resource surfaces, use ID3D10Device::CopySubresourceRegion. For resizing operations, render to a texture by using texture filtering. For converting MSAA surfaces to non-MSAA surfaces, use ID3D10Device::ResolveSubresource.

Additional Direct3D 10.1 Differences

Windows Vista with Service Pack 1 (SP1) included a minor update to Direct3D 10 and Direct3D 10.1, which exposed the following additional hardware features:

- MSAA per-sample shaders
- MSAA depth read-back
- Independent blend modes per render target
- Cube map arrays
- Render to block-compressed (BC) formats

The Direct3D 10.1 update added support for the following new interfaces, which are derived from existing interfaces:

- ID3D10Device1
- ID3D10BlendState1
- ID3D10ShaderResourceView1

The Direct3D 10.1 update also included the following additional structures:

- D3D10_RENDER_TARGET_BLEND_DESC1
- D3D10_BLEND_DESC1
- D3D10_SHADER_RESOURCE_VIEW_DESC1

The Direct3D 10.1 API includes a new concept named feature level. This concept means that you can use the Direct3D 10.1 API to drive Direct3D 10.0 (D3D10_FEATURE_LEVEL_10_0) or Direct3D 10.1 (D3D10_FEATURE_LEVEL_10_1) hardware. Because the Direct3D 10.1 API is derived from the Direct3D 10 interfaces, applications can create a Direct3D 10.1 device, then use it as a Direct3D 10.0 device except where new 10.1-specific features are needed (provided that the **D3D10_FEATURE_LEVEL_10_1** feature-level is present and supports these features).

> **Note**
>
> Direct3D 10.1 devices can use the existing 10.0 HLSL shader profiles (vs_4_0, ps_4_0, gs_4_0) and the new 10.1 profiles (vs_4_1, ps_4_1, gs_4_1) with additional HLSL

instructions and capabilities.

Windows 7 contained a minor update to the Direct3D 10.1 API that is included in the Direct3D 11 runtime. This update adds support for the following feature levels:

- **D3D10_FEATURE_LEVEL_9_1**
- **D3D10_FEATURE_LEVEL_9_2**
- **D3D10_FEATURE_LEVEL_9_3**

Windows 7 also added support to Direct3D 10.1 for Windows Advanced Rasterization Platform (WARP). You can specify a WARP driver by using **D3D10_DRIVER_TYPE_WARP**.

For more information about Direct3D 10.1, see Direct3D 10.1 Features and the **D3D10_FEATURE_LEVEL1** enumeration.

Related topics

Programming Guide for Direct3D 10

Linking Static and Dynamic Libraries (Direct3D 10)

Article • 01/06/2021 • 2 minutes to read

For an application to run properly, the host computer must have the appropriate DLLs installed. These DLLs may be provided by either the operating system, or the applications' redistributable package.

Libraries Load Appropriate DLLs

The libraries included with the DirectX SDK will automatically load the proper DLLs at runtime. The exception to this rule is d3dx10.lib/d3dx10d.lib, which will load the d3dx10.dll that was shipped with that version of the SDK. For example, if the downloaded SDK includes d3dx10_33.dll and d3dx10_34.dll, then the library (d3dx10.lib) that shipped with that SDK will load d3dx10_34.dll. If a subsequent SDK is installed later containing d3dx10_35.lib, the d3dx10.lib from the previous SDK will still load d3dx10_34.dll. The d3dx10.lib from the newer SDK will load d3dx10_35.dll.

Redistributing Binaries

Only d3dx10.dll (and subsequent versions of the same file) can be redistributed. To redistribute this file, you must use the **DirectXSetup** function. For details on using this function and putting together a redistributable package, see Installing DirectX with DirectSetup. All other necessary binaries are included in Windows Vista. The only binaries that can be redistributed are those located in the following directory.

```
(SDK root)\Redist
```

The following table describes the binaries developers should be aware of.

Direct3D 10 Binaries	Description
d3dx10.dll/d3dx10d.dll	Retail and debug D3DX10 components; the retail components can be redistributed in the REDIST CAB.

Direct3D 10 Binaries	Description
d3d10ref.dll	Reference Rasterizer. Provides software implementation of the graphics pipeline. Only included as part of the Windows SDK or legacy DirectX SDK and cannot be redistributed. The Reference Rasterizer is intended for debugging only. Explicit linking is not necessary; attempting to create a reference device (see **D3D10CreateDevice**) will load this dll if it is present.
d3d10sdklayers.dll	A series of SDK utilities that act as a layer between API calls and runtime execution, including the debug layer and the switch-to-reference layer. Explicit linking is not necessary; if a device is created with the appropriate layer flag, this DLL is loaded automatically. This component is meant for development and debugging purposes only. Only included as part of the Windows SDK or legacy DirectX SDK and cannot be redistributed.

Related topics

Programming Guide for Direct3D 10

Direct3D 10 Graphics

Direct3D 10.1 Features

Article • 01/06/2021 • 4 minutes to read

Direct3D 10.1 extends the feature set of Direct3D 10.0 with the following new features:

- Blend Modes - Independent blend modes per render target using the new blend-state interface (see **ID3D10BlendState1 Interface**). Dual source blending operations are restricted to render target slot 0; you may not write to other outputs or have any render targets bound to slots other than slot 0.

- Culling Behavior - Zero-area faces are automatically culled; this affects wireframe rendering only.

- Floating Point Rules - Uses the same IEEE-754 rules for floating-point EXCEPT 32-bit floating point operations have been tightened to produce a result within 0.5 unit-last-place (0.5 ULP) of the infinitely precise result. This applies to addition, subtraction, and multiplication. (accuracy to 0.5 ULP for multiply, 1.0 ULP for reciprocal).

- Formats - The precision of float16 blending has increased to 0.5 ULP. Blending is also required for UNORM16/SNORM16/SNORM8 formats.

- Multisample Anti-Aliasing - Multisampling has been enhanced to generalize coverage based transparency and make multisampling work more effectively with multi-pass rendering. To achieve this, all multisample semantics are defined as if the pixel shader always runs once per sample (sample-frequency), computing a separate color per sample. If a pixel shader doesn't use any per-sample attributes, then it will compute the same value for each covered sample in a pixel. In that case, it is equivalent to the hardware executing the shader once per pixel (pixel-frequency), replicating the result to all covered samples. Naturally, running at pixel-frequency always produces the same results as running the same shader at sample-frequency, when the attributes are sampled at a pixel-frequency. The PSInvocations pipeline statistic increments at sample-frequency unless the shader is running at pixel-frequency.

- Pipeline Stage Bandwidth - Increased the amount of data that can be passed between shader stages:

Resource	Limits
Registers between Shader Stages	32 (32-bit x 4-component)
Vertex Shader Input Registers	32

Resource	Limits
Input Assembler input slots	32

- Rasterization Rules - The rules for rasterization have changed for lines, in addition, new functionality has been added.
  - MultisampleEnable only affects line rasterization (points and triangles are unaffected), and is used to choose a line drawing algorithm. This means that some multisample rasterization from Direct3D 10 are no longer supported.
  - New sample-frequency pixel shader execution with primitive rasterization.
- Resources - CopyResource is enabled in two new scenarios:
  - Both color and depth/stencil MSAA surfaces can now be used with CopyResource as either a source or destination
  - Format Conversion while copying between certain 32/64/128 bit prestructured, typed resources and compressed representations of the same bit widths.
- Texture Sampling - sample_c and sample_c_lz instructions are defined to work with both Texture2DArrays and TextureCubeArrays, use the Location member (the alpha component) to specify an an array index.
- Views - TextureCube and the new TextureCubeArray (see **D3D10_TEXCUBE_ARRAY_SRV1**) are not actual resources, but are new views on a Texture2DArray resource. Create a resource view from a Texture2DArray resource with a new usage flag (D3D10_RESOURCE_MISC_TEXTURECUBE), use the new **ID3D10ShaderResourceView1 Interface** interface to bind a cube-texture view to the pipeline.

The new features require a 10.1 device type (see **ID3D10Device1 Interface**) which can be created by calling **D3D10CreateDevice1**, or you can create the device and swap chain at the same time by calling **D3D10CreateDeviceAndSwapChain1**.

In Windows Vista Service Pack 1, Direct3D 10.0 and Direct3D 10.1 DLLs exist side-by-side on the system. To access 10.1 features, do either of the following:

Accessing 10.1 Features on Vista Gold and Vista Service Pack 1

Developers that wish to support Vista Gold as well as SP1 will have to account for the lack of the new 10.1 API extensions on Vista Gold. Both DXUT and D3DX10 will provide

convenience functions to create the appropriate device, based on the DLLs available on the system and the available hardware (10.0 or 10.1). The 10.1 device inherits from the 10.0 device, and can be retrieved using QueryInterface(). It is recommended that each application keeps track of the device type and maintains a pointer to the 10.1 device (if available) to avoid frequent QueryInterface calls when 10.1 functionality is desired. Likewise, where 10.1 resource views and state objects are associated by an application's custom class, it is recommended that the application track whether the object is a 10.0 or 10.1 type to avoid redundant QueryInterface() calls. D3DX10 includes a set of utility functions to simplify this process (see **D3DX10CreateDevice** and **D3DX10CreateDeviceAndSwapChain**).

Accessing 10.1 Features on Vista Service Pack 1 Exclusively

Some developers may choose to require Vista Service Pack 1, which will be distributed broadly to end-users and includes a series of improvements outside of Direct3D 10.1. These developers can use the Direct3D 10.1 headers and libraries exclusively, taking a dependency on the Direct3D 10.1 DLLs which support both 10.0 and 10.1 hardware (some calls may fail, however, on 10.0 devices where the new functionality is not supported).

Some additional notes:

- The APIs exposed in the D3DX10.dll will accept both 10.0 and 10.1 devices, and will take advantage of 10.1 functionality when available.
- D3D10SDKLayers.dll supports a 10.1 device and can output the appropriate debug spew for 10.1 features.
- D3D10Ref.dll implements a 10.0 and 10.1 software device.
- D3DX10 and FXC support the updated 10.1 shader model with the following targets: vs_4_1, gs_4_1, ps_4_1, and fx_4_1 which can be bound to a 10.1 device. A 10.1 device supports shader model 4.0 and 4.1 shaders.
- The Direct3D 10.0 effect framework supports 10.0 and 10.1 devices, however, any technique that includes shader model 4.1 shaders or the new 10.1 features must use a 10.1 device.

Related topics

Programming Guide for Direct3D 10

Reference for Direct3D 10

Article • 01/06/2021 • 2 minutes to read

The Direct3D 10 API is described in the following sections:

- Direct3D Reference
- D3DX Reference

Related topics

Direct3D 10 Graphics

Direct3D Reference (Direct3D 10 Graphics)

Article • 06/19/2021 • 2 minutes to read

This section contains the following sections:

- Core Reference
- Resource Reference
- Shader Reference
- Effect Reference
- Direct3D 10 Return Codes

Related topics

Reference for Direct3D 10

Direct3D 10 core reference

Article • 06/19/2021 • 2 minutes to read

The Direct3D API defines several core API elements.

- Core Interfaces
- Core Functions
- Core Structures
- Core Enumerations

Related topics

Direct3D Reference

Direct3D 10 core interfaces

Article • 01/06/2021 • 2 minutes to read

This section contains information about the following core interfaces:

Interfaces	Description
ID3D10BlendState Interface	Accesses blending state for a Direct3D 10.0 device.
ID3D10BlendState1 Interface	Accesses blending state for a Direct3D 10.1 device.
ID3D10DepthStencilState Interface	Accesses depth-stencil state.
ID3D10InputLayout Interface	Accesses pipeline-input data from memory.
ID3D10RasterizerState Interface	Accesses rasterizer state.
ID3D10SamplerState Interface	Accesses sampler state.

Interfaces	Description
ID3D10Asynchronous Interface	Retrieves data from the GPU asynchronously.
ID3D10Blob Interface	Return data from memory.
ID3D10Counter Interface	Measures GPU performance.
ID3D10Debug Interface	Enables/disables the debug layer.
ID3D10Device Interface	Represents a virtual adapter for Direct3D 10.0.
ID3D10Device1 Interface	Represents a virtual adapter for Direct3D 10.1.
ID3D10DeviceChild Interface	Accesses data used by a device.
ID3D10Include Interface	Provides user-overridable methods for handling include files when loading an effect.
ID3D10InfoQueue Interface	Stores, retrieves, and filters debug messages.
ID3D10Multithread Interface	Accesses multithread settings.

Interfaces	Description
ID3D10Predicate Interface	Determines whether geometry should be processed.
ID3D10Query Interface	Queries information from the GPU.
ID3D10StateBlock Interface	Encapsulates render states.
ID3D10SwitchToRef Interface	Switches between a hardware and software device.

Direct3D also impements interfaces for:

- Resources
- Shaders
- Effects

Related topics

Core Reference

ID3D10Asynchronous interface (d3d10.h)

Article07/27/2022

This interface encapsulates methods for retrieving data from the GPU asynchronously.

Inheritance

The **ID3D10Asynchronous** interface inherits from ID3D10DeviceChild.
ID3D10Asynchronous also has these types of members:

Methods

The **ID3D10Asynchronous** interface has these methods.

ID3D10Asynchronous::Begin Starts the collection of GPU data.
ID3D10Asynchronous::End Ends the collection of GPU data.
ID3D10Asynchronous::GetData Get data from the GPU asynchronously.
ID3D10Asynchronous::GetDataSize Get the size of the data (in bytes) that is output when calling ID3D10Asynchronous::GetData.

Remarks

There are three types of asynchronous interfaces, all of which inherit this interface:

- ID3D10Query Interface - Queries information from the GPU.
- ID3D10Predicate Interface - Determines whether a piece of geometry should be processed or not depending on the results of a previous draw call.
- ID3D10Counter Interface - Measures GPU performance.

Requirements

Target Platform	Windows
Header	d3d10.h

See also

Core Interfaces

ID3D10DeviceChild

ID3D10Asynchronous::Begin method (d3d10.h)

Article • 06/29/20212 minutes to read

Starts the collection of GPU data.

Syntax

```C++
void Begin();
```

Return value

None

Remarks

Calling Begin starts the asynchronous collection of GPU data. Calling ID3D10Asynchronous::End causes data collection to stop.
See ID3D10Asynchronous Interface for additional information.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Asynchronous Interface

ID3D10Asynchronous::End method (d3d10.h)

Article06/29/2021

Ends the collection of GPU data.

Syntax

```
C++

void End();
```

Return value

None

Remarks

Calling ID3D10Asynchronous::Begin starts the asynchronous collection of GPU data. Calling **ID3D10Asynchronous::End** causes data collection to stop. See ID3D10Asynchronous Interface for additional information.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Asynchronous Interface

ID3D10Asynchronous::GetData method (d3d10.h)

Article10/13/2021

Get data from the GPU asynchronously.

Syntax

```
C++

HRESULT GetData(
  [out] void *pData,
  [in]  UINT DataSize,
  [in]  UINT GetDataFlags
);
```

Parameters

`[out] pData`

Type: **void***

Address of memory that will receive the data. If **NULL**, **GetData** will be used only to check status. The type of data output depends on the type of asynchronous interface. See Remarks.

`[in] DataSize`

Type: **UINT**

Size of the data to retrieve or 0. This value can be obtained with ID3D10Asynchronous::GetDataSize. Must be 0 when *pData* is **NULL**.

`[in] GetDataFlags`

Type: **UINT**

Optional flags. Can be 0 or any combination of the flags enumerated by D3D10_ASYNC_GETDATA_FLAG.

Return value

Type: **HRESULT**

If this function succeeds, it returns S_OK. Otherwise, possible return values are the following:

- S_FALSE
- DXGI_ERROR_DEVICE_REMOVED
- DXGI_ERROR_INVALID_CALL

Remarks

GetData retrieves the data collected between calls to ID3D10Asynchronous::Begin and ID3D10Asynchronous::End. Certain queries only require a call to **ID3D10Asynchronous::End** in which case the data returned by **GetData** is accurate up to the last call to **ID3D10Asynchronous::End** (See ID3D10Query Interface).

If *DataSize* is 0, **GetData** is only used to check status where a return value of S_OK indicates that data is available to give to an application, and a return value of S_FALSE indicates data is not yet available.

It is invalid to invoke this function on a predicate created with the flag D3D10_QUERY_MISCFLAG_PREDICATEHINT.

If the asynchronous interface that calls this function is ID3D10Query Interface, then the following table applies.

Query Type	Output Data Type	Supports Begin Method
D3D10_QUERY_EVENT	BOOL	NO
D3D10_QUERY_OCCLUSION	UINT64	YES
D3D10_QUERY_TIMESTAMP	UINT64	NO
D3D10_QUERY_TIMESTAMP_DISJOINT	D3D10_QUERY_DATA_TIMESTAMP_DISJOINT	YES
D3D10_QUERY_PIPELINE_STATISTICS	D3D10_QUERY_DATA_PIPELINE_STATISTICS	YES
D3D10_QUERY_OCCLUSION_PREDICATE	BOOL	YES
D3D10_QUERY_SO_STATISTICS	D3D10_QUERY_DATA_SO_STATISTICS	YES
D3D10_QUERY_SO_OVERFLOW_PREDICATE	BOOL	YES

If the asynchronous interface that calls this API is ID3D10Counter Interface, then the following applies.

Counter Type	Output Data Type	Units
D3D10_COUNTER_GPU_IDLE	FLOAT32	fraction of time
D3D10_COUNTER_VERTEX_PROCESSING	FLOAT32	fraction of time
D3D10_COUNTER_GEOMETRY_PROCESSING	FLOAT32	fraction of time
D3D10_COUNTER_PIXEL_PROCESSING	FLOAT32	fraction of time
D3D10_COUNTER_OTHER_GPU_PROCESSING	FLOAT32	fraction of time
D3D10_COUNTER_HOST_ADAPTER_BANDWIDTH_UTILIZATION	FLOAT32	fraction of theoretical maximum
D3D10_COUNTER_LOCAL_VIDMEM_BANDWIDTH_UTILIZATION	FLOAT32	fraction of theoretical maximum
D3D10_COUNTER_VERTEX_THROUGHPUT_UTILIZATION	FLOAT32	fraction of theoretical maximum
D3D10_COUNTER_TRIANGLE_SETUP_THROUGHPUT_UTILIZATION	FLOAT32	fraction of theoretical maximum
D3D10_COUNTER_FILLRATE_THROUGHPUT_UTILIZATION	FLOAT32	fraction of theoretical maximum
D3D10_COUNTER_VS_MEMORY_LIMITED	FLOAT32	fraction of time
D3D10_COUNTER_VS_COMPUTATION_LIMITED	FLOAT32	fraction of time
D3D10_COUNTER_GS_MEMORY_LIMITED	FLOAT32	fraction of time
D3D10_COUNTER_GS_COMPUTATION_LIMITED	FLOAT32	fraction of time
D3D10_COUNTER_PS_MEMORY_LIMITED	FLOAT32	fraction of time
D3D10_COUNTER_PS_COMPUTATION_LIMITED	FLOAT32	fraction of time
D3D10_COUNTER_POST_TRANSFORM_CACHE_HIT_RATE	FLOAT32	fraction
D3D10_COUNTER_TEXTURE_CACHE_HIT_RATE	FLOAT32	fraction

The value returned by a D3D10_COUNTER_GPU_IDLE, D3D10_COUNTER_VERTEX_PROCESSING, D3D10_COUNTER_GEOMETRY_PROCESSING, D3D10_COUNTER_PIXEL_PROCESSING, or D3D10_COUNTER_OTHER_GPU_PROCESSING counter may be different depending on the number of parallel counters that exist on a video card, and those values can be interpreted with the following equation:

$$\frac{(\text{VERTEX_PROCESSING} + \text{GEOMETRY_PROCESSING} + \text{PIXEL_PROCESSING} + \text{OTHER_GPU_PROCESSING})}{\text{NumDetectableParallelUnits}} + \text{GPU_IDLE} = 100\% \text{ of time}$$

The number of parallel counters that a video card has is available from **NumDetectableParallelUnits** in D3D10_COUNTER_INFO, and it can be retrieved by calling ID3D10Device::CheckCounterInfo.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Asynchronous Interface

ID3D10Asynchronous::GetDataSize method (d3d10.h)

Article • 06/29/20212 minutes to read

Get the size of the data (in bytes) that is output when calling ID3D10Asynchronous::GetData.

Syntax

```
C++

UINT GetDataSize();
```

Return value

Type: **UINT**

Size of the data (in bytes) that is output when calling GetData.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Asynchronous Interface

ID3D10BlendState interface (d3d10.h)

Article • 07/27/20222 minutes to read

This blend-state interface accesses blending state for a Direct3D 10.0 device for the output-merger stage.

Inheritance

The **ID3D10BlendState** interface inherits from ID3D10DeviceChild. **ID3D10BlendState** also has these types of members:

Methods

The **ID3D10BlendState** interface has these methods.

ID3D10BlendState::GetDesc Get the blend state. (ID3D10BlendState.GetDesc)

Remarks

Blending combines two pixel values. You have control over how the pixels are blended by using a predefined set of blending operations, as well as preblending operations. The Blending Block Diagram shows conceptually how blending works.

To create a blend-state interface, call ID3D10Device::CreateBlendState. To initialize the blend state, call ID3D10Device::OMSetBlendState.

Requirements

Target Platform	Windows
Header	d3d10.h

See also

Core Interfaces

ID3D10DeviceChild

ID3D10BlendState::GetDesc method (d3d10.h)

Article • 07/27/20222 minutes to read

Get the blend state.

Syntax

```
C++

void GetDesc(
  [out] D3D10_BLEND_DESC *pDesc
);
```

Parameters

`[out] pDesc`

Type: **D3D10_BLEND_DESC***

A pointer to the blend state (see D3D10_BLEND_DESC).

Return value

None

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10BlendState Interface

ID3D10BlendState1 interface (d3d10_1.h)

Article • 07/27/20222 minutes to read

This blend-state interface accesses blending state for a Direct3D 10.1 device for the output-merger stage.

Inheritance

The **ID3D10BlendState1** interface inherits from ID3D10BlendState. **ID3D10BlendState1** also has these types of members:

Methods

The **ID3D10BlendState1** interface has these methods.

ID3D10BlendState1::GetDesc1 Get the blend state. (ID3D10BlendState1.GetDesc1)

Remarks

Blending combines two pixel values. You have control over how the pixels are blended by using a predefined set of blending operations, as well as preblending operations. The Blending Block Diagram shows conceptually how blending works.

To create a blend-state interface, call ID3D10Device1::CreateBlendState1. To initialize the blend state, call ID3D10Device::OMSetBlendState.

This method requires Windows Vista Service Pack 1.

Requirements

Target Platform	Windows
Header	d3d10_1.h

See also

Core Interfaces

ID3D10BlendState

ID3D10Blob interface (d3dcommon.h)

Article • 07/22/20212 minutes to read

This interface is used to return arbitrary-length data.

Inheritance

The **ID3D10Blob** interface inherits from the IUnknown interface. **ID3D10Blob** also has these types of members:

Methods

The **ID3D10Blob** interface has these methods.

ID3D10Blob::GetBufferPointer Gets a pointer to the data.
ID3D10Blob::GetBufferSize Gets the size.

Remarks

The ID3DBlob interface is type-defined in the D3DCommon.h header file as a **ID3D10Blob** interface, which is fully defined in the D3DCommon.h header file. **ID3DBlob** is version-neutral and can be used in code for any Direct3D version.

Blobs can be used as a data buffer, storing vertex, adjacency, and material information during mesh optimization and loading operations. Also, these objects are used to return object code and error messages in APIs that compile vertex, geometry and pixel shaders.

Requirements

Target Platform	Windows
Header	d3dcommon.h

See also

Common Version Interfaces

ID3DBlob

IUnknown

ID3D10Blob::GetBufferPointer method (d3dcommon.h)

Article06/29/2021

Gets a pointer to the data.

Syntax

```
C++

LPVOID GetBufferPointer();
```

Return value

Type: **LPVOID**

Returns a pointer.

Requirements

Target Platform	Windows
Header	d3dcommon.h

See also

ID3D10Blob

ID3D10Blob::GetBufferSize method (d3dcommon.h)

Article06/29/2021

Gets the size.

Syntax

```
C++

SIZE_T GetBufferSize();
```

Return value

Type: **SIZE_T**

The size of the data, in bytes.

Requirements

Target Platform	Windows
Header	d3dcommon.h

See also

ID3D10Blob

ID3D10Counter interface (d3d10.h)

Article • 07/27/20222 minutes to read

This interface encapsulates methods for measuring GPU performance.

Inheritance

The **ID3D10Counter** interface inherits from ID3D10Asynchronous. **ID3D10Counter** also has these types of members:

Methods

The **ID3D10Counter** interface has these methods.

ID3D10Counter::GetDesc Get a counter description. (ID3D10Counter.GetDesc)

Remarks

A counter can be created with ID3D10Device::CreateCounter.

This is a derived class of ID3D10Asynchronous Interface.

Counter data is gathered by issuing an ID3D10Asynchronous::Begin command, issuing some graphics commands, issuing an ID3D10Asynchronous::End command, and then calling ID3D10Asynchronous::GetData to get data about what happened in between the Begin and End calls. The data returned by GetData will be different depending on the type of counter. The call to End causes the data returned by GetData to be accurate up until the last call to End.

Counters are best suited for profiling.

For a list of the types of performance counters, see D3D10_COUNTER.

Requirements

Target Platform	Windows
Header	d3d10.h

See also

Core Interfaces

ID3D10Asynchronous

ID3D10Counter::GetDesc method (d3d10.h)

Article • 07/27/20222 minutes to read

Get a counter description.

Syntax

```
C++

void GetDesc(
  [out] D3D10_COUNTER_DESC *pDesc
);
```

Parameters

`[out] pDesc`

Type: **D3D10_COUNTER_DESC***

Pointer to a counter description (see D3D10_COUNTER_DESC).

Return value

None

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Counter Interface

ID3D10Debug interface (d3d10sdklayers.h)

Article • 07/27/20222 minutes to read

A debug interface controls debug settings, validates pipeline state and can only be used if the debug layer is turned on.

Inheritance

The **ID3D10Debug** interface inherits from the IUnknown interface. **ID3D10Debug** also has these types of members:

Methods

The **ID3D10Debug** interface has these methods.

ID3D10Debug::GetFeatureMask Get a bitfield of flags that indicates which debug features are on or off. (ID3D10Debug.GetFeatureMask)
ID3D10Debug::GetPresentPerRenderOpDelay Get the number of milliseconds to sleep after Present is called.
ID3D10Debug::GetSwapChain Get the swap chain that the runtime will use for automatically calling Present.
ID3D10Debug::SetFeatureMask Set a bitfield of flags that will turn debug features on and off.
ID3D10Debug::SetPresentPerRenderOpDelay Set the number of milliseconds to sleep after Present is called.
ID3D10Debug::SetSwapChain Set a swap chain that the runtime will use for automatically calling Present.

ID3D10Debug::Validate Check the validity of pipeline state.	

Remarks

This interface is obtained by querying it from the ID3D10Device Interface using IUnknown::QueryInterface.

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

Core Interfaces

ID3D10Debug::GetFeatureMask method (d3d10sdklayers.h)

Article • 07/27/20222 minutes to read

Get a bitfield of flags that indicates which debug features are on or off.

Syntax

```
C++

UINT GetFeatureMask();
```

Return value

Type: **UINT**

Mask of feature-mask flags bitwise ORed together. If a flag is present, then that feature will be set to on, otherwise the feature will be set to off. See ID3D10Debug::SetFeatureMask for a list of possible feature-mask flags.

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10Debug Interface

ID3D10Debug::GetPresentPerRenderOpDelay method (d3d10sdklayers.h)

Article • 06/29/20212 minutes to read

Get the number of milliseconds to sleep after Present is called.

Syntax

```
C++

UINT GetPresentPerRenderOpDelay();
```

Return value

Type: **UINT**

Number of milliseconds to sleep after Present is called.

Remarks

Value is set with ID3D10Debug::SetPresentPerRenderOpDelay.

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10Debug Interface

ID3D10Debug::GetSwapChain method (d3d10sdklayers.h)

Article • 10/13/20212 minutes to read

Get the swap chain that the runtime will use for automatically calling Present.

Syntax

```
C++

HRESULT GetSwapChain(
  [out] IDXGISwapChain **ppSwapChain
);
```

Parameters

`[out] ppSwapChain`

Type: **IDXGISwapChain****

Swap chain that the runtime will use for automatically calling Present.

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

The swap chain retrieved by this method will only be used if D3D10_DEBUG_FEATURE_PRESENT_PER_RENDER_OP is set in the feature mask.

Requirements

Target Platform	Windows

Header	d3d10sdklayers.h

See also

ID3D10Debug Interface

ID3D10Debug::SetFeatureMask method (d3d10sdklayers.h)

Article • 10/13/20212 minutes to read

Set a bitfield of flags that will turn debug features on and off.

Syntax

```
C++

HRESULT SetFeatureMask(
  [in] UINT Mask
);
```

Parameters

`[in] Mask`

Type: **UINT**

Feature-mask flags bitwise ORed together. If a flag is present, then that feature will be set to on, otherwise the feature will be set to off. See remarks for a list of flags.

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

> **Note** If you call this API in a Session 0 process, it returns **DXGI_ERROR_NOT_CURRENTLY_AVAILABLE**.

Setting a feature-mask flag will cause a rendering-operation method (listed below) to do some extra task when called. The possible feature flags are:

D3D10_DEBUG_FEATURE_FINISH_PER_RENDER_OP	Application will wait for the GPU to finish

	processing the rendering operation before continuing.
D3D10_DEBUG_FEATURE_FLUSH_PER_RENDER_OP	Runtime will additionally call ID3D10Device::Flush.
D3D10_DEBUG_FEATURE_PRESENT_PER_RENDER_OP	Runtime will call Present. Presentation of render buffers will occur according to the settings established by prior calls to ID3D10Debug::SetSwapChain and ID3D10Debug::SetPresentPerRenderOpDelay.

These feature-mask flags apply to the following rendering-operation methods:

- ID3D10Device::Draw
- ID3D10Device::DrawIndexed
- ID3D10Device::DrawInstanced
- ID3D10Device::DrawIndexedInstanced
- ID3D10Device::DrawAuto
- ID3D10Device::ClearRenderTargetView
- ID3D10Device::ClearDepthStencilView
- ID3D10Device::CopySubresourceRegion
- ID3D10Device::CopyResource
- ID3D10Device::UpdateSubresource
- ID3D10Device::GenerateMips
- ID3D10Device::ResolveSubresource

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10Debug Interface

ID3D10Debug::SetPresentPerRenderOpDelay method (d3d10sdklayers.h)

Article • 10/13/20212 minutes to read

Set the number of milliseconds to sleep after Present is called.

Syntax

```
C++

HRESULT SetPresentPerRenderOpDelay(
  [in] UINT Milliseconds
);
```

Parameters

`[in] Milliseconds`

Type: **UINT**

Number of milliseconds to sleep after Present is called.

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

> **Note** If you call this API in a Session 0 process, it returns **DXGI_ERROR_NOT_CURRENTLY_AVAILABLE**.

The application will only sleep if D3D10_DEBUG_FEATURE_PRESENT_PER_RENDER_OP is a set in the feature mask. If that flag is not set the number of milliseconds is set but ignored and the application does not sleep. 10ms is used as a default value if this method is never called.

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10Debug Interface

ID3D10Debug::SetSwapChain method (d3d10sdklayers.h)

Article • 10/13/20212 minutes to read

Set a swap chain that the runtime will use for automatically calling Present.

Syntax

```
C++

HRESULT SetSwapChain(
  [in] IDXGISwapChain *pSwapChain
);
```

Parameters

`[in] pSwapChain`

Type: **IDXGISwapChain***

Swap chain that the runtime will use for automatically calling Present; must have been created with the DXGI_SWAP_EFFECT_SEQUENTIAL swap-effect flag.

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

> **Note** If you call this API in a Session 0 process, it returns **DXGI_ERROR_NOT_CURRENTLY_AVAILABLE**.

The swap chain set by this method will only be used if D3D10_DEBUG_FEATURE_PRESENT_PER_RENDER_OP is set in the feature mask.

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10Debug Interface

ID3D10Debug::Validate method (d3d10sdklayers.h)

Article • 06/29/20212 minutes to read

Check the validity of pipeline state.

Syntax

```
C++

HRESULT Validate();
```

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

When the debug layer is turned on all draw functions will do this operation.

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10Debug Interface

ID3D10DepthStencilState interface (d3d10.h)

Article • 07/22/20212 minutes to read

A depth-stencil-state interface accesses depth-stencil state which sets up the depth-stencil test for the output-merger stage.

Inheritance

The **ID3D10DepthStencilState** interface inherits from ID3D10DeviceChild. **ID3D10DepthStencilState** also has these types of members:

Methods

The **ID3D10DepthStencilState** interface has these methods.

ID3D10DepthStencilState::GetDesc Get the depth-stencil state.

Remarks

Create a depth-stencil state object by calling ID3D10Device::CreateDepthStencilState.

To initialize depth-stencil state, bind the depth-stencil-state object to the pipeline by calling ID3D10Device::OMSetDepthStencilState.

Requirements

Target Platform	Windows
Header	d3d10.h

See also

Core Interfaces

ID3D10DeviceChild

ID3D10DepthStencilState::GetDesc method (d3d10.h)

Article10/13/2021

Get the depth-stencil state.

Syntax

```
C++

void GetDesc(
  [out] D3D10_DEPTH_STENCIL_DESC *pDesc
);
```

Parameters

`[out] pDesc`

Type: **D3D10_DEPTH_STENCIL_DESC***

A pointer to the depth-stencil state (see D3D10_DEPTH_STENCIL_DESC).

Return value

None

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10DepthStencilState Interface

ID3D10Device interface (d3d10.h)

Article07/27/2022

The device interface represents a virtual adapter for Direct3D 10.0; it is used to perform rendering and create Direct3D resources.

Inheritance

The **ID3D10Device** interface inherits from the IUnknown interface. **ID3D10Device** also has these types of members:

Methods

The **ID3D10Device** interface has these methods.

ID3D10Device::CheckCounter Get the type, name, units of measure, and a description of an existing counter. (ID3D10Device.CheckCounter)
ID3D10Device::CheckCounterInfo Get a counter's information. (ID3D10Device.CheckCounterInfo)
ID3D10Device::CheckFormatSupport Get the support of a given format on the installed video device. (ID3D10Device.CheckFormatSupport)
ID3D10Device::CheckMultisampleQualityLevels Get the number of quality levels available during multisampling. (ID3D10Device.CheckMultisampleQualityLevels)
ID3D10Device::ClearDepthStencilView Clears the depth-stencil resource. (ID3D10Device.ClearDepthStencilView)
ID3D10Device::ClearRenderTargetView Set all the elements in a render target to one value. (ID3D10Device.ClearRenderTargetView)

ID3D10Device::ClearState Restore all default device settings; return the device to the state it was in when it was created.
ID3D10Device::CopyResource Copy the entire contents of the source resource to the destination resource using the GPU. (ID3D10Device.CopyResource)
ID3D10Device::CopySubresourceRegion Copy a region from a source resource to a destination resource. (ID3D10Device.CopySubresourceRegion)
ID3D10Device::CreateBlendState Create a blend-state object that encapsulates blend state for the output-merger stage. (ID3D10Device.CreateBlendState)
ID3D10Device::CreateBuffer Create a buffer (vertex buffer, index buffer, or shader-constant buffer).
ID3D10Device::CreateCounter Create a counter object for measuring GPU performance. (ID3D10Device.CreateCounter)
ID3D10Device::CreateDepthStencilState Create a depth-stencil state object that encapsulates depth-stencil test information for the output-merger stage. (ID3D10Device.CreateDepthStencilState)
ID3D10Device::CreateDepthStencilView Create a depth-stencil view for accessing resource data. (ID3D10Device.CreateDepthStencilView)
ID3D10Device::CreateGeometryShader Create a geometry shader. (ID3D10Device.CreateGeometryShader)
ID3D10Device::CreateGeometryShaderWithStreamOutput Creates a geometry shader that can write to streaming output buffers. (ID3D10Device.CreateGeometryShaderWithStreamOutput)
ID3D10Device::CreateInputLayout Create an input-layout object to describe the input-buffer data for the input-assembler stage. (ID3D10Device.CreateInputLayout)

ID3D10Device::CreatePixelShader Create a pixel shader. (ID3D10Device.CreatePixelShader)
ID3D10Device::CreatePredicate Creates a predicate. (ID3D10Device.CreatePredicate)
ID3D10Device::CreateQuery This interface encapsulates methods for querying information from the GPU. (ID3D10Device.CreateQuery)
ID3D10Device::CreateRasterizerState Create a rasterizer state object that tells the rasterizer stage how to behave. (ID3D10Device.CreateRasterizerState)
ID3D10Device::CreateRenderTargetView Create a render-target view for accessing resource data.
ID3D10Device::CreateSamplerState Create a sampler-state object that encapsulates sampling information for a texture. (ID3D10Device.CreateSamplerState)
ID3D10Device::CreateShaderResourceView Create a shader-resource view for accessing data in a resource. (ID3D10Device.CreateShaderResourceView)
ID3D10Device::CreateTexture1D Create an array of 1D textures (see Texture1D).
ID3D10Device::CreateTexture2D Create an array of 2D textures (see Texture2D).
ID3D10Device::CreateTexture3D Create a single 3D texture (see Texture3D).
ID3D10Device::CreateVertexShader Create a vertex-shader object from a compiled shader. (ID3D10Device.CreateVertexShader)

ID3D10Device::Draw	Draw non-indexed, non-instanced primitives. (ID3D10Device.Draw)
ID3D10Device::DrawAuto	Draw geometry of an unknown size that was created by the geometry shader stage. See remarks.
ID3D10Device::DrawIndexed	Draw indexed, non-instanced primitives. (ID3D10Device.DrawIndexed)
ID3D10Device::DrawIndexedInstanced	Draw indexed, instanced primitives. (ID3D10Device.DrawIndexedInstanced)
ID3D10Device::DrawInstanced	Draw non-indexed, instanced primitives. (ID3D10Device.DrawInstanced)
ID3D10Device::Flush	Send queued-up commands in the command buffer to the GPU.
ID3D10Device::GenerateMips	Generates mipmaps for the given shader resource. (ID3D10Device.GenerateMips)
ID3D10Device::GetCreationFlags	Get the flags used during the call to create the device with D3D10CreateDevice.
ID3D10Device::GetDeviceRemovedReason	Get the reason why the device was removed. (ID3D10Device.GetDeviceRemovedReason)
ID3D10Device::GetExceptionMode	Get the exception-mode flags. (ID3D10Device.GetExceptionMode)
ID3D10Device::GetPredication	Get the rendering predicate state. (ID3D10Device.GetPredication)
ID3D10Device::GetPrivateData	Get data from a device that is associated with a guid.

ID3D10Device::GetTextFilterSize This method is not implemented. (ID3D10Device.GetTextFilterSize)
ID3D10Device::GSGetConstantBuffers Get the constant buffers used by the geometry shader pipeline stage. (ID3D10Device.GSGetConstantBuffers)
ID3D10Device::GSGetSamplers Get an array of sampler states from the geometry shader pipeline stage.
ID3D10Device::GSGetShader Get the geometry shader currently set on the device. (ID3D10Device.GSGetShader)
ID3D10Device::GSGetShaderResources Get the geometry shader resources. (ID3D10Device.GSGetShaderResources)
ID3D10Device::GSSetConstantBuffers Set the constant buffers used by the geometry shader pipeline stage.
ID3D10Device::GSSetSamplers Set an array of sampler states to the geometry shader pipeline stage. (ID3D10Device.GSSetSamplers)
ID3D10Device::GSSetShader Set a geometry shader to the device. (ID3D10Device.GSSetShader)
ID3D10Device::GSSetShaderResources Bind an array of shader resources to the geometry shader stage. (ID3D10Device.GSSetShaderResources)
ID3D10Device::IAGetIndexBuffer Get a pointer to the index buffer that is bound to the input-assembler stage. (ID3D10Device.IAGetIndexBuffer)
ID3D10Device::IAGetInputLayout Get a pointer to the input-layout object that is bound to the input-assembler stage. (ID3D10Device.IAGetInputLayout)

ID3D10Device::IAGetPrimitiveTopology Get information about the primitive type, and data order that describes input data for the input assembler stage. (ID3D10Device.IAGetPrimitiveTopology)
ID3D10Device::IAGetVertexBuffers Get the vertex buffers bound to the input-assembler stage. (ID3D10Device.IAGetVertexBuffers)
ID3D10Device::IASetIndexBuffer Bind an index buffer to the input-assembler stage. (ID3D10Device.IASetIndexBuffer)
ID3D10Device::IASetInputLayout Bind an input-layout object to the input-assembler stage. (ID3D10Device.IASetInputLayout)
ID3D10Device::IASetPrimitiveTopology Bind information about the primitive type, and data order that describes input data for the input assembler stage. (ID3D10Device.IASetPrimitiveTopology)
ID3D10Device::IASetVertexBuffers Bind an array of vertex buffers to the input-assembler stage. (ID3D10Device.IASetVertexBuffers)
ID3D10Device::OMGetBlendState Get the blend state of the output-merger stage. (ID3D10Device.OMGetBlendState)
ID3D10Device::OMGetDepthStencilState Gets the depth-stencil state of the output-merger stage. (ID3D10Device.OMGetDepthStencilState)
ID3D10Device::OMGetRenderTargets Get pointers to the render targets and the depth-stencil buffer that are available to the output-merger stage.
ID3D10Device::OMSetBlendState Set the blend state of the output-merger stage. (ID3D10Device.OMSetBlendState)
ID3D10Device::OMSetDepthStencilState Sets the depth-stencil state of the output-merger stage. (ID3D10Device.OMSetDepthStencilState)

ID3D10Device::OMSetRenderTargets	Bind one or more render targets and the depth-stencil buffer to the output-merger stage.
ID3D10Device::OpenSharedResource	Give a device access to a shared resource created on a different Direct3d device.
ID3D10Device::PSGetConstantBuffers	Get the constant buffers used by the pixel shader pipeline stage. (ID3D10Device.PSGetConstantBuffers)
ID3D10Device::PSGetSamplers	Get an array of sampler states from the pixel shader pipeline stage. (ID3D10Device.PSGetSamplers)
ID3D10Device::PSGetShader	Get the pixel shader currently set on the device. (ID3D10Device.PSGetShader)
ID3D10Device::PSGetShaderResources	Get the pixel shader resources. (ID3D10Device.PSGetShaderResources)
ID3D10Device::PSSetConstantBuffers	Set the constant buffers used by the pixel shader pipeline stage.
ID3D10Device::PSSetSamplers	Set an array of sampler states to the pixel shader pipeline stage. (ID3D10Device.PSSetSamplers)
ID3D10Device::PSSetShader	Sets a pixel shader to the device. (ID3D10Device.PSSetShader)
ID3D10Device::PSSetShaderResources	Bind an array of shader resources to the pixel shader stage. (ID3D10Device.PSSetShaderResources)
ID3D10Device::ResolveSubresource	Copy a multisampled resource into a non-multisampled resource. This API is most useful when re-using the resulting rendertarget of one render pass as an input to a second render pass.

ID3D10Device::RSGetScissorRects Get the array of scissor rectangles bound to the rasterizer stage. (ID3D10Device.RSGetScissorRects)
ID3D10Device::RSGetState Get the rasterizer state from the rasterizer stage of the pipeline. (ID3D10Device.RSGetState)
ID3D10Device::RSGetViewports Get the array of viewports bound to the rasterizer stage
ID3D10Device::RSSetScissorRects Bind an array of scissor rectangles to the rasterizer stage. (ID3D10Device.RSSetScissorRects)
ID3D10Device::RSSetState Set the rasterizer state for the rasterizer stage of the pipeline. (ID3D10Device.RSSetState)
ID3D10Device::RSSetViewports Bind an array of viewports to the rasterizer stage of the pipeline. (ID3D10Device.RSSetViewports)
ID3D10Device::SetExceptionMode Get the exception-mode flags. (ID3D10Device.SetExceptionMode)
ID3D10Device::SetPredication Set a rendering predicate. (ID3D10Device.SetPredication)
ID3D10Device::SetPrivateData Set data to a device and associate that data with a guid. (ID3D10Device.SetPrivateData)
ID3D10Device::SetPrivateDataInterface Associate an IUnknown-derived interface with this device and associate that interface with an application-defined guid.
ID3D10Device::SetTextFilterSize This method is not implemented. (ID3D10Device.SetTextFilterSize)
ID3D10Device::SOGetTargets Get the target output buffers for the StreamOutput stage of the pipeline.

ID3D10Device::SOSetTargets	Set the target output buffers for the StreamOutput stage, which enables/disables the pipeline to stream-out data.
ID3D10Device::UpdateSubresource	The CPU copies data from memory to a subresource created in non-mappable memory. See remarks.
ID3D10Device::VSGetConstantBuffers	Get the constant buffers used by the vertex shader pipeline stage. (ID3D10Device.VSGetConstantBuffers)
ID3D10Device::VSGetSamplers	Get an array of sampler states from the vertex shader pipeline stage. (ID3D10Device.VSGetSamplers)
ID3D10Device::VSGetShader	Get the vertex shader currently set on the device. (ID3D10Device.VSGetShader)
ID3D10Device::VSGetShaderResources	Get the vertex shader resources. (ID3D10Device.VSGetShaderResources)
ID3D10Device::VSSetConstantBuffers	Set the constant buffers used by the vertex shader pipeline stage.
ID3D10Device::VSSetSamplers	Set an array of sampler states to the vertex shader pipeline stage. (ID3D10Device.VSSetSamplers)
ID3D10Device::VSSetShader	Set a vertex shader to the device. (ID3D10Device.VSSetShader)
ID3D10Device::VSSetShaderResources	Bind an array of shader resources to the vertex shader stage.

Remarks

A device is created using D3D10CreateDevice.

Requirements

Target Platform	Windows
Header	d3d10.h

See also

Core Interfaces

ID3D10Device::CheckCounter method (d3d10.h)

Article07/27/2022

Get the type, name, units of measure, and a description of an existing counter.

Syntax

```
C++

HRESULT CheckCounter(
  [in]      const D3D10_COUNTER_DESC *pDesc,
  [out]     D3D10_COUNTER_TYPE       *pType,
  [out]     UINT                     *pActiveCounters,
  [out]     LPSTR                    szName,
  [in, out] UINT                     *pNameLength,
  [out]     LPSTR                    szUnits,
  [in, out] UINT                     *pUnitsLength,
  [out]     LPSTR                    szDescription,
  [in, out] UINT                     *pDescriptionLength
);
```

Parameters

`[in] pDesc`

Type: **const** **D3D10_COUNTER_DESC***

Pointer to a counter description (see D3D10_COUNTER_DESC). Specifies which counter information is to be retrieved about.

`[out] pType`

Type: **D3D10_COUNTER_TYPE***

Pointer to the data type of a counter (see D3D10_COUNTER_TYPE). Specifies the data type of the counter being retrieved.

`[out] pActiveCounters`

Type: **UINT***

Pointer to the number of hardware counters that are needed for this counter type to be created. All instances of the same counter type use the same hardware counters.

`[out] szName`

Type: **LPSTR**

String to be filled with a brief name for the counter. May be **NULL** if the application is not interested in the name of the counter.

`[in, out] pNameLength`

Type: **UINT***

Length of the string returned to szName. Can be **NULL**.

`[out] szUnits`

Type: **LPSTR**

Name of the units a counter measures, provided the memory the pointer points to has enough room to hold the string. Can be **NULL**. The returned string will always be in English.

`[in, out] pUnitsLength`

Type: **UINT***

Length of the string returned to szUnits. Can be **NULL**.

`[out] szDescription`

Type: **LPSTR**

A description of the counter, provided the memory the pointer points to has enough room to hold the string. Can be **NULL**. The returned string will always be in English.

`[in, out] pDescriptionLength`

Type: **UINT***

Length of the string returned to szDescription. Can be **NULL**.

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

Length parameters can be **NULL**, which indicates the application is not interested in the length nor the corresponding string value. When a length parameter is non-**NULL** and the corresponding string is **NULL**, the input value of the length parameter is ignored, and the length of the corresponding string (including terminating **NULL**) will be returned through the length parameter. When length and the corresponding parameter are both non-**NULL**, the input value of length is checked to ensure there is enough room, and then the length of the string (including terminating **NULL** character) is passed out through the length parameter.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::CheckCounterInfo method (d3d10.h)

Article07/27/2022

Get a counter's information.

Syntax

```
C++

void CheckCounterInfo(
  [out] D3D10_COUNTER_INFO *pCounterInfo
);
```

Parameters

`[out] pCounterInfo`

Type: **D3D10_COUNTER_INFO***

Pointer to counter information (see D3D10_COUNTER_INFO).

Return value

None

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::CheckFormatSupport method (d3d10.h)

Article07/27/2022

Get the support of a given format on the installed video device.

Syntax

C++

```
HRESULT CheckFormatSupport(
  [in]  DXGI_FORMAT Format,
  [out] UINT        *pFormatSupport
);
```

Parameters

`[in] Format`

Type: **DXGI_FORMAT**

A DXGI_FORMAT enumeration that describes a format for which to check for support.

`[out] pFormatSupport`

Type: **UINT***

A bitfield of D3D10_FORMAT_SUPPORT enumeration values describing how the specified format is supported on the installed device. The values are ORed together.

Return value

Type: **HRESULT**

Returns S_OK if successful; otherwise, returns E_INVALIDARG if the *Format* parameter is **NULL**, or returns E_FAIL if the described format does not exist.

Remarks

Most format support is based on the Direct3D feature level. Only a few specific use cases require checking for support. See Hardware Support for Direct3D 10 Formats and Hardware Support for Direct3D 10.1 Formats for additional information.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::CheckMultisampleQualityLevels method (d3d10.h)

Article • 07/27/20222 minutes to read

Get the number of quality levels available during multisampling.

Syntax

C++

```
HRESULT CheckMultisampleQualityLevels(
  [in]  DXGI_FORMAT Format,
  [in]  UINT        SampleCount,
  [out] UINT        *pNumQualityLevels
);
```

Parameters

`[in] Format`

Type: **DXGI_FORMAT**

The texture format. See DXGI_FORMAT.

`[in] SampleCount`

Type: **UINT**

The number of samples during multisampling.

`[out] pNumQualityLevels`

Type: **UINT***

Number of quality levels supported by the adapter. See remarks.

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

When multisampling a texture, the number of quality levels available for an adapter is dependent on the texture format used and the number of samples requested. The maximum sample count defined by D3D10_MAX_MULTISAMPLE_SAMPLE_COUNT in d3d10.h is 32. If the returned value of *pNumQualityLevels* is 0, the format and sample count combination is not supported for the installed adapter.

Furthermore, the definition of a quality level is up to each hardware vendor to define, however no facility is provided by Direct3D to help discover this information.

Direct3D 10.1 devices are required to support 4x MSAA for all formats except R32G32B32A32 and R32G32B32 formats.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::ClearDepthStencilView method (d3d10.h)

Article • 07/27/20222 minutes to read

Clears the depth-stencil resource.

Syntax

C++

```
void ClearDepthStencilView(
  [in] ID3D10DepthStencilView *pDepthStencilView,
  [in] UINT                   ClearFlags,
  [in] FLOAT                  Depth,
  [in] UINT8                  Stencil
);
```

Parameters

`[in] pDepthStencilView`

Type: **ID3D10DepthStencilView***

Pointer to the depth stencil to be cleared.

`[in] ClearFlags`

Type: **UINT**

Which parts of the buffer to clear. See D3D10_CLEAR_FLAG.

`[in] Depth`

Type: **FLOAT**

Clear the depth buffer with this value. This value will be clamped between 0 and 1.

`[in] Stencil`

Type: **UINT8**

Clear the stencil buffer with this value.

Return value

None

Remarks

Differences between Direct3D 9 and Direct3D 10:
Unlike Direct3D 9, the full extent of the resource view is always cleared. Viewport and scissor settings are not applied.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::ClearRenderTargetView method (d3d10.h)

Article07/27/2022

Set all the elements in a render target to one value.

Syntax

```
C++

void ClearRenderTargetView(
  [in] ID3D10RenderTargetView *pRenderTargetView,
  [in] const FLOAT [4]        ColorRGBA
);
```

Parameters

`[in] pRenderTargetView`

Type: **ID3D10RenderTargetView***

Pointer to the render target.

`[in] ColorRGBA`

Type: **const FLOAT**

A 4-component array that represents the color to fill the render target with.

Return value

None

Remarks

Applications that wish to clear a render target to a specific integer value bit pattern should render a screen-aligned quad instead of using this method. The reason for this is because this method accepts as input a floating point value, which may not have the same bit pattern as the original integer.

Differences between Direct3D 9 and Direct3D 10:
Unlike Direct3D 9, the full extent of the resource view is always cleared. Viewport and scissor settings are not applied.

When using 10Level9, **ClearRenderTargetView** only clears the first array slice in the render target view. This can impact (for example) cube map rendering scenarios. Applications should create a render target view for each face or array slice, then clear each view individually.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::ClearState method (d3d10.h)

Article06/29/2021

Restore all default device settings; return the device to the state it was in when it was created. This will set all set all input/output resource slots, shaders, input layouts, predications, scissor rectangles, depth-stencil state, rasterizer state, blend state, sampler state, and viewports to **NULL**. The primitive topology will be set to UNDEFINED.

Syntax

```C++
void ClearState();
```

Return value

None

Requirements

Target Platform	Windows
Header	d3d10.h (include D3d10core)

See also

ID3D10Device Interface

ID3D10Device::CopyResource method (d3d10.h)

Article07/27/2022

Copy the entire contents of the source resource to the destination resource using the GPU.

Syntax

```
C++

void CopyResource(
  [in] ID3D10Resource *pDstResource,
  [in] ID3D10Resource *pSrcResource
);
```

Parameters

`[in] pDstResource`

Type: **ID3D10Resource***

A pointer to the destination resource (see ID3D10Resource).

`[in] pSrcResource`

Type: **ID3D10Resource***

A pointer to the source resource (see ID3D10Resource).

Return value

None

Remarks

This method is unusual in that it causes the GPU to perform the copy operation (similar to a memcpy by the CPU). As a result, it has a few restrictions designed for improving performance. For instance, the source and destination resources:

- Must be different resources.
- Must be the same type.
- Must have identical dimensions (including width, height, depth, and size as appropriate).
- Must have compatible formats, which means the formats must be identical or at least from the same type group. For example, a DXGI_FORMAT_R32G32B32_FLOAT texture can be copied to a DXGI_FORMAT_R32G32B32_UINT texture since both of these formats are in the DXGI_FORMAT_R32G32B32_TYPELESS group. Beginning with Direct3D 10.1, **CopyResource** can copy between a few format types. For more info, see Format Conversion using Direct3D 10.1.
- May not be currently mapped.

CopyResource supports only copy; it does not support any stretch, color key, blend, or format conversions. Beginning with Direct3D 10.1, **CopyResource** can reinterpret the resource data between a few format types. For more info, see Format Conversion using Direct3D 10.1.

Immutable, and depth-stencil resources cannot be used as a destination. Resources created with multisampling capability cannot be used as either a source or destination.

The method is an asynchronous call which may be added to the command-buffer queue. This attempts to remove pipeline stalls that may occur when copying data. See performance considerations for more details.

An application that only needs to copy a portion of the data in a resource should use ID3D10Device::CopySubresourceRegion instead.

> Differences between Direct3D 10 and Direct3D 10.1:
> Direct3D 10.1 enables depth-stencil resources to be used as either a source or destination.
> Direct3D 10.1 enables multisampled resources to be used as source and destination only if both source and destination have identical multisampled count and quality. If source and destination differ in multisampled count and quality or one of them is multisampled and the other is not multisampled, the call to **ID3D10Device::CopyResource** fails.
>
> It is possible to copy between prestructured+typed resources and block-compressed textures. See Format Conversion using Direct3D 10.1.

Requirements

Target Platform	Windows
Header	d3d10.h

Library	D3D10.lib

See also

ID3D10Device

ID3D10Resource Interface

ID3D10Device::CopySubresourceRegion method (d3d10.h)

Article • 07/27/20224 minutes to read

Copy a region from a source resource to a destination resource.

Syntax

C++

```
void CopySubresourceRegion(
  [in] ID3D10Resource  *pDstResource,
  [in] UINT            DstSubresource,
  [in] UINT            DstX,
  [in] UINT            DstY,
  [in] UINT            DstZ,
  [in] ID3D10Resource  *pSrcResource,
  [in] UINT            SrcSubresource,
  [in] const D3D10_BOX *pSrcBox
);
```

Parameters

`[in] pDstResource`

Type: **ID3D10Resource***

A pointer to the destination resource (see ID3D10Resource).

`[in] DstSubresource`

Type: **UINT**

Subresource index of the destination.

`[in] DstX`

Type: **UINT**

The x coordinate of the upper left corner of the destination region.

`[in] DstY`

Type: **UINT**

The y coordinate of the upper left corner of the destination region.

`[in] DstZ`

Type: **UINT**

The z coordinate of the upper left corner of the destination region. For a 1D or 2D subresource, this must be zero.

`[in] pSrcResource`

Type: **ID3D10Resource***

A pointer to the source resource (see ID3D10Resource).

`[in] SrcSubresource`

Type: **UINT**

Subresource index of the source.

`[in] pSrcBox`

Type: **const D3D10_BOX***

A 3D box (see D3D10_BOX) that defines the source subresource that can be copied. If **NULL**, the entire source subresource is copied. The box must fit within the source resource.

An empty box results in a no-op. A box is empty if the top value is greater than or equal to the bottom value, or the left value is greater than or equal to the right value, or the front value is greater than or equal to the back value. When the box is empty, **CopySubresourceRegion** doesn't perform a copy operation.

Return value

None

Remarks

The source box must be within the size of the source resource. The destination location is an absolute value (not a relative value). The destination location can be offset from the source location; however, the size of the region to copy (including the destination location) must fit in the destination resource.

If the resources are buffers, all coordinates are in bytes; if the resources are textures, all coordinates are in texels.

D3D10CalcSubresource is a helper function for calculating subresource indexes.

CopySubresourceRegion performs the copy on the GPU (similar to a memcpy by the CPU). As a consequence, the source and destination resources must meet the following criteria:

- Must be different subresources (although they can be from the same resource).
- Must be the same type.
- Must have compatible formats (the formats must either be identical or be from the same type group). For example, a DXGI_FORMAT_R32G32B32_FLOAT texture can be copied to a DXGI_FORMAT_R32G32B32_UINT texture because both of these formats are in the DXGI_FORMAT_R32G32B32_TYPELESS group. Beginning with Direct3D 10.1, **CopySubresourceRegion** can copy between a few format types. For more info, see Format Conversion using Direct3D 10.1.
- May not be currently mapped.

CopySubresourceRegion supports only copy; it does not support any stretch, color key, blend, or format conversions. Beginning with Direct3D 10.1, **CopySubresourceRegion** can reinterpret the resource data between a few format types. For more info, see Format Conversion using Direct3D 10.1.

If your app needs to copy an entire resource, we recommend to use ID3D10Device::CopyResource instead.

CopySubresourceRegion is an asynchronous call that the runtime can add to the command-buffer queue. This asynchronous behaviorattempts to remove pipeline stalls that may occur when copying data. See performance considerations for more details.

Differences between Direct3D 10 and Direct3D 10.1:
Direct3D 10 has the following limitations:

- You cannot use a depth-stencil resource as a destination.
- You cannot use an immutable resource as a destination.
- You cannot use a multisampled texture as either a source or a destination

Direct3D 10.1 has added support for the following features:

- You can use a depth-stencil buffer as a source or a destination.
- You can use multisampled resources as source and destination only if both source and destination have identical multisampled count and quality. If source and destination differ in multisampled count and quality or if the source is multisampled and the destination is not multisampled (or vice versa), the call to **ID3D10Device::CopySubresourceRegion** fails.

- You can copy between uncompressed and compressed resources. During copy, the format conversions that are specified in Format Conversion using Direct3D 10.1 are supported automatically. The uncompressed resource must be at least prestructured, and typed. You must also account for the difference between the virtual and the physical size of the mipmaps levels.

Note If you use **CopySubresourceRegion** with a depth-stencil buffer or a multisampled resource, you must copy the whole subresource. You must also pass 0 to the *DstX*, *DstY*, and *DstZ* parameters and **NULL** to the *pSrcBox* parameter. In addition, source and destination resources, which are represented by the *pSrcResource* and *pDstResource* parameters respectively, must have identical sample count values.

Example

The following code snippet copies a box (located at (120,100),(200,220)) from a source texture into a region (130,120),(210,240) in a destination texture.

```
D3D10_BOX sourceRegion;
sourceRegion.left = 120;
sourceRegion.right = 200;
sourceRegion.top = 100;
sourceRegion.bottom = 220;
sourceRegion.front = 0;
sourceRegion.back = 1;

pd3dDevice->CopySubresourceRegion( pDestTexture, 0, 130, 120, 0,
pSourceTexture, 0, &sourceRegion );
```

Notice that, for a 2D texture, front and back are always set to 0 and 1 respectively.

Requirements

Target Platform	Windows

Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device

ID3D10Resource Interface

ID3D10Device::CreateBlendState method (d3d10.h)

Article • 07/27/20222 minutes to read

Create a blend-state object that encapsulates blend state for the output-merger stage.

Syntax

C++

```
HRESULT CreateBlendState(
  [in]  const D3D10_BLEND_DESC *pBlendStateDesc,
  [out] ID3D10BlendState       **ppBlendState
);
```

Parameters

`[in] pBlendStateDesc`

Type: **const D3D10_BLEND_DESC***

Pointer to a blend-state description (see D3D10_BLEND_DESC).

`[out] ppBlendState`

Type: **ID3D10BlendState****

Address of a pointer to the blend-state object created (see ID3D10BlendState Interface).

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

An application can create up to 4096 unique blend-state objects. For each object created, the runtime checks to see if a previous object has the same state. If such a

previous object exists, the runtime will return a pointer to previous instance instead of creating a duplicate object.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::CreateBuffer method (d3d10.h)

Article • 10/13/20212 minutes to read

Create a buffer (vertex buffer, index buffer, or shader-constant buffer).

Syntax

C++

```
HRESULT CreateBuffer(
  [in]  const D3D10_BUFFER_DESC      *pDesc,
  [in]  const D3D10_SUBRESOURCE_DATA *pInitialData,
  [out] ID3D10Buffer                 **ppBuffer
);
```

Parameters

`[in] pDesc`

Type: **const D3D10_BUFFER_DESC***

Pointer to a buffer description (see D3D10_BUFFER_DESC).

`[in] pInitialData`

Type: **const D3D10_SUBRESOURCE_DATA***

Pointer to the initialization data (see D3D10_SUBRESOURCE_DATA); use **NULL** to allocate space only.

`[out] ppBuffer`

Type: **ID3D10Buffer****

Address of a pointer to the buffer created (see ID3D10Buffer Interface). Set this parameter to **NULL** to validate the other input parameters (S_FALSE indicates a pass).

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

For example code, see:

- Create a Vertex Buffer
- Create an Index Buffer

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::CreateCounter method (d3d10.h)

Article • 07/27/20222 minutes to read

Create a counter object for measuring GPU performance.

Syntax

C++

```
HRESULT CreateCounter(
  [in]  const D3D10_COUNTER_DESC *pCounterDesc,
  [out] ID3D10Counter            **ppCounter
);
```

Parameters

`[in] pCounterDesc`

Type: **const D3D10_COUNTER_DESC***

Pointer to a counter description (see D3D10_COUNTER_DESC).

`[out] ppCounter`

Type: **ID3D10Counter****

Address of a pointer to a counter (see ID3D10Counter Interface).

Return value

Type: **HRESULT**

If this function succeeds, it will return S_OK. If it fails, possible return values are: S_FALSE, E_OUTOFMEMORY, DXGI_ERROR_UNSUPPORTED, DXGI_ERROR_NONEXCLUSIVE, or E_INVALIDARG.

DXGI_ERROR_UNSUPPORTED is returned whenever the application requests to create a well-known counter, but the current device does not support it.

DXGI_ERROR_NONEXCLUSIVE indicates that another device object is currently using the counters, so they cannot be used by this device at the moment.

E_INVALIDARG is returned whenever an out-of-range well-known or device-dependent counter is requested, or when the simulataneously active counters have been exhausted.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::CreateDepthStencilState method (d3d10.h)

Article • 07/27/20222 minutes to read

Create a depth-stencil state object that encapsulates depth-stencil test information for the output-merger stage.

Syntax

```
C++

HRESULT CreateDepthStencilState(
  [in]  const D3D10_DEPTH_STENCIL_DESC *pDepthStencilDesc,
  [out] ID3D10DepthStencilState        **ppDepthStencilState
);
```

Parameters

`[in] pDepthStencilDesc`

Type: **const D3D10_DEPTH_STENCIL_DESC***

Pointer to a depth-stencil state description (see D3D10_DEPTH_STENCIL_DESC).

`[out] ppDepthStencilState`

Type: **ID3D10DepthStencilState****

Address of a pointer to the depth-stencil state object created (see ID3D10DepthStencilState Interface).

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

4096 unique depth-stencil state objects can be created on a device at a time.

If an application attempts to create a depth-stencil state with the same description as an already existing depth-stencil state, then the same interface with an incremented reference count will be returned and the total number of unique depth-stencil state objects will stay the same.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::CreateDepthStencilView method (d3d10.h)

Article • 07/27/20222 minutes to read

Create a depth-stencil view for accessing resource data.

Syntax

C++

```
HRESULT CreateDepthStencilView(
  [in]  ID3D10Resource                      *pResource,
  [in]  const D3D10_DEPTH_STENCIL_VIEW_DESC *pDesc,
  [out] ID3D10DepthStencilView              **ppDepthStencilView
);
```

Parameters

`[in] pResource`

Type: **ID3D10Resource***

Pointer to the resource that will serve as the depth-stencil surface. This resource must have been created with the D3D10_BIND_DEPTH_STENCIL flag.

`[in] pDesc`

Type: **const D3D10_DEPTH_STENCIL_VIEW_DESC***

Pointer to a depth-stencil-view description (see D3D10_DEPTH_STENCIL_VIEW_DESC). Set this parameter to **NULL** to create a view that accesses mipmap level 0 of the entire resource (using the format the resource was created with).

`[out] ppDepthStencilView`

Type: **ID3D10DepthStencilView****

Address of a pointer to an ID3D10DepthStencilView. Set this parameter to **NULL** to validate the other input parameters (the method will return S_FALSE if the other input parameters pass validation).

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

A depth-stencil view can be bound to the output-merger stage by calling ID3D10Device::OMSetRenderTargets.

For more background information, see the programming guide page about depth stencils.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::CreateGeometryShader method (d3d10.h)

Article • 07/27/20222 minutes to read

Create a geometry shader.

Syntax

C++

```
HRESULT CreateGeometryShader(
  [in]  const void           *pShaderBytecode,
  [in]  SIZE_T               BytecodeLength,
  [out] ID3D10GeometryShader **ppGeometryShader
);
```

Parameters

`[in] pShaderBytecode`

Type: **const void***

A pointer to the compiled shader. To get this pointer see Getting a Pointer to a Compiled Shader.

`[in] BytecodeLength`

Type: **SIZE_T**

Size of the compiled geometry shader.

`[out] ppGeometryShader`

Type: **ID3D10GeometryShader****

Address of a pointer to an ID3D10GeometryShader Interface. If this is **NULL**, all other parameters will be validated, and if all parameters pass validation this API will return S_FALSE instead of S_OK.

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

Once created, the shader can be set to the device by calling ID3D10Device::GSSetShader.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::CreateGeometryShaderWithStreamOutput method (d3d10.h)

Article • 07/27/20222 minutes to read

Creates a geometry shader that can write to streaming output buffers.

Syntax

C++

```
HRESULT CreateGeometryShaderWithStreamOutput(
  [in]  const void                      *pShaderBytecode,
  [in]  SIZE_T                          BytecodeLength,
  [in]  const D3D10_SO_DECLARATION_ENTRY *pSODeclaration,
  [in]  UINT                            NumEntries,
  [in]  UINT                            OutputStreamStride,
  [out] ID3D10GeometryShader            **ppGeometryShader
);
```

Parameters

`[in] pShaderBytecode`

Type: **const void***

A pointer to the compiled geometry shader for a standard geometry shader plus stream output. For info on how to get this pointer, see Getting a Pointer to a Compiled Shader.

To create the stream output without using a geometry shader, pass a pointer to the output signature for the prior stage. To obtain this output signature, call the D3DGetOutputSignatureBlob compiler function. You can also pass a pointer to the compiled vertex shader that is used in the prior stage. This compiled shader provides the output signature for the data.

`[in] BytecodeLength`

Type: **SIZE_T**

Size of the compiled geometry shader.

`[in] pSODeclaration`

Type: **const D3D10_SO_DECLARATION_ENTRY***

Pointer to a D3D10_SO_DECLARATION_ENTRY array. Cannot be **NULL** if *NumEntries*> 0.

`[in] NumEntries`

Type: **UINT**

The number of entries in the array pointed to by *pSODeclaration*. Minimum 0, maximum 64.

`[in] OutputStreamStride`

Type: **UINT**

The size, in bytes, of each element in the array pointed to by *pSODeclaration*. This parameter is only used when the output slot is 0 for all entries in *pSODeclaration*.

`[out] ppGeometryShader`

Type: **ID3D10GeometryShader****

Address of a pointer to an ID3D10GeometryShader Interface. If this is **NULL**, all other parameters will be validated, and if all parameters pass validation this API will return S_FALSE instead of S_OK.

Return value

Type: **HRESULT**

This method returns one of the Direct3D 10 Return Codes.

Remarks

For more info about using **CreateGeometryShaderWithStreamOutput**, see Create a Geometry-Shader Object with Stream Output.

Requirements

Target Platform	Windows
Header	d3d10.h

Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::CreateInputLayout method (d3d10.h)

Article • 07/27/20222 minutes to read

Create an input-layout object to describe the input-buffer data for the input-assembler stage.

Syntax

```
C++

HRESULT CreateInputLayout(
  [in]  const D3D10_INPUT_ELEMENT_DESC *pInputElementDescs,
  [in]  UINT                           NumElements,
  [in]  const void                     *pShaderBytecodeWithInputSignature,
  [in]  SIZE_T                         BytecodeLength,
  [out] ID3D10InputLayout              **ppInputLayout
);
```

Parameters

`[in] pInputElementDescs`

Type: **const D3D10_INPUT_ELEMENT_DESC***

An array of the input-assembler stage input data types; each type is described by an element description (see D3D10_INPUT_ELEMENT_DESC).

`[in] NumElements`

Type: **UINT**

The number of input-data types in the array of input-elements.

`[in] pShaderBytecodeWithInputSignature`

Type: **const void***

A pointer to the compiled shader. To get this pointer see Getting a Pointer to a Compiled Shader. The compiled shader code contains a input signature which is validated against the array of elements. See remarks.

`[in] BytecodeLength`

Type: **SIZE_T**

Size of the compiled shader.

`[out] ppInputLayout`

Type: **ID3D10InputLayout****

A pointer to the input-layout object created (see ID3D10InputLayout Interface). To validate the other input parameters, set this pointer to be **NULL** and verify that the method returns S_FALSE.

Return value

Type: **HRESULT**

If the method succeeds, the return code is S_OK. See Direct3D 10 Return Codes for failing error codes.

Remarks

After creating an input layout object, it must be bound to the input-assembler stage before calling a draw API. See Getting Started with the Input-Assembler Stage (Direct3D 10) for example code.

Once an input-layout object is created from a shader signature, the input-layout object can be reused with any other shader that has an identical input signature (semantics included). This can simplify the creation of input-layout objects when you are working with many shaders with identical inputs.

If a data type in the input-layout declaration does not match the data type in a shader-input signature, CreateInputLayout will generate a warning during compilation. The warning is simply to call attention to the fact that the data may be reinterpreted when read from a register. You may either disregard this warning (if reinterpretation is intentional) or make the data types match in both declarations to eliminate the warning. The Data Conversion Rules overview describes the rules applied for data type conversion.

> Differences between Direct3D 9 and Direct3D 10:
> Mapping the vertex data to the shader inputs with an input layout is a new way of doing things in Direct3D 10 that improves performance.

In Direct3D 10 the vertex data is mapped to the shader inputs when the input layout object is created, whereas in Direct3D 9 this mapping was done at Draw time based on the currently bound vertex declarations, vertex buffers, and vertex shaders. Doing this mapping when the input layout object is created reduces or eliminates extra linkage work for drivers at Draw time because this re-mapping is no longer necessary.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::CreatePixelShader method (d3d10.h)

Article • 07/27/20222 minutes to read

Create a pixel shader.

Syntax

C++

```
HRESULT CreatePixelShader(
  [in]  const void        *pShaderBytecode,
  [in]  SIZE_T            BytecodeLength,
  [out] ID3D10PixelShader **ppPixelShader
);
```

Parameters

`[in] pShaderBytecode`

Type: **const void***

A pointer to the compiled shader. To get this pointer see Getting a Pointer to a Compiled Shader.

`[in] BytecodeLength`

Type: **SIZE_T**

Size of the compiled pixel shader.

`[out] ppPixelShader`

Type: **ID3D10PixelShader****

Address of a pointer to an ID3D10PixelShader Interface. If this is **NULL**, all other parameters will be validated, and if all parameters pass validation this API will return S_FALSE instead of S_OK.

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

After creating the pixel shader, you can set it to the device using ID3D10Device::PSSetShader.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::CreatePredicate method (d3d10.h)

Article07/27/2022

Creates a predicate.

Syntax

```
C++

HRESULT CreatePredicate(
  [in]  const D3D10_QUERY_DESC *pPredicateDesc,
  [out] ID3D10Predicate        **ppPredicate
);
```

Parameters

`[in] pPredicateDesc`

Type: **const D3D10_QUERY_DESC***

Pointer to a query description where the type of query must be a D3D10_QUERY_SO_OVERFLOW_PREDICATE or D3D10_QUERY_OCCLUSION_PREDICATE (see D3D10_QUERY_DESC).

`[out] ppPredicate`

Type: **ID3D10Predicate****

Address of a pointer to a predicate (see ID3D10Predicate Interface).

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::CreateQuery method (d3d10.h)

Article • 07/27/20222 minutes to read

This interface encapsulates methods for querying information from the GPU.

Syntax

```
C++

HRESULT CreateQuery(
  [in]  const D3D10_QUERY_DESC *pQueryDesc,
  [out] ID3D10Query            **ppQuery
);
```

Parameters

`[in] pQueryDesc`

Type: **const D3D10_QUERY_DESC***

Pointer to a query description (see D3D10_QUERY_DESC).

`[out] ppQuery`

Type: **ID3D10Query****

Address of a pointer to the query object created (see ID3D10Query Interface).

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows

Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::CreateRasterizerState method (d3d10.h)

Article07/27/2022

Create a rasterizer state object that tells the rasterizer stage how to behave.

Syntax

```
C++

HRESULT CreateRasterizerState(
  [in]  const D3D10_RASTERIZER_DESC *pRasterizerDesc,
  [out] ID3D10RasterizerState       **ppRasterizerState
);
```

Parameters

`[in] pRasterizerDesc`

Type: **const D3D10_RASTERIZER_DESC***

Pointer to a rasterizer state description (see D3D10_RASTERIZER_DESC).

`[out] ppRasterizerState`

Type: **ID3D10RasterizerState****

Address of a pointer to the rasterizer state object created (see ID3D10RasterizerState Interface).

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

4096 unique rasterizer state objects can be created on a device at a time.

If an application attempts to create a rasterizer state with the same description as an already existing rasterizer state, then the same interface with an incremented reference count will be returned and the total number of unique rasterizer state objects will stay the same.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::CreateRenderTargetView method (d3d10.h)

Article • 10/13/20212 minutes to read

Create a render-target view for accessing resource data.

Syntax

```
C++

HRESULT CreateRenderTargetView(
  [in]  ID3D10Resource                      *pResource,
  [in]  const D3D10_RENDER_TARGET_VIEW_DESC *pDesc,
  [out] ID3D10RenderTargetView              **ppRTView
);
```

Parameters

`[in] pResource`

Type: **ID3D10Resource***

Pointer to the resource that will serve as the render target. This resource must have been created with the D3D10_BIND_RENDER_TARGET flag.

`[in] pDesc`

Type: **const D3D10_RENDER_TARGET_VIEW_DESC***

Pointer to a render-target-view description (see D3D10_RENDER_TARGET_VIEW_DESC). Set this parameter to **NULL** to create a view that accesses mipmap level 0 of the entire resource (using the format the resource was created with).

`[out] ppRTView`

Type: **ID3D10RenderTargetView****

Address of a pointer to an ID3D10RenderTargetView. Set this parameter to **NULL** to validate the other input parameters (the method will return S_FALSE if the other input parameters pass validation).

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

A rendertarget view can be bound to the output merger stage by calling ID3D10Device::OMSetRenderTargets.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::CreateSamplerState method (d3d10.h)

Article • 07/27/20222 minutes to read

Create a sampler-state object that encapsulates sampling information for a texture.

Syntax

```
C++

HRESULT CreateSamplerState(
  [in]  const D3D10_SAMPLER_DESC *pSamplerDesc,
  [out] ID3D10SamplerState       **ppSamplerState
);
```

Parameters

`[in] pSamplerDesc`

Type: **const D3D10_SAMPLER_DESC***

Pointer to a sampler state description (see D3D10_SAMPLER_DESC).

`[out] ppSamplerState`

Type: **ID3D10SamplerState****

Address of a pointer to the sampler state object created (see ID3D10SamplerState Interface).

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

4096 unique sampler state objects can be created on a device at a time.

If an application attempts to create a sampler state with the same description as an already existing sampler state, then the same interface with an incremented reference count will be returned and the total number of unique sampler state objects will stay the same.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::CreateShaderResourceView method (d3d10.h)

Article • 07/27/20222 minutes to read

Create a shader-resource view for accessing data in a resource.

Syntax

```
C++

HRESULT CreateShaderResourceView(
  [in]  ID3D10Resource                        *pResource,
  [in]  const D3D10_SHADER_RESOURCE_VIEW_DESC *pDesc,
  [out] ID3D10ShaderResourceView              **ppSRView
);
```

Parameters

`[in] pResource`

Type: **ID3D10Resource***

Pointer to the resource that will serve as input to a shader. This resource must have been created with the D3D10_BIND_SHADER_RESOURCE flag.

`[in] pDesc`

Type: **const D3D10_SHADER_RESOURCE_VIEW_DESC***

Pointer to a shader-resource-view description (see D3D10_SHADER_RESOURCE_VIEW_DESC). Set this parameter to **NULL** to create a view that accesses the entire resource (using the format the resource was created with).

`[out] ppSRView`

Type: **ID3D10ShaderResourceView****

Address of a pointer to an ID3D10ShaderResourceView. Set this parameter to **NULL** to validate the other input parameters (the method will return S_FALSE if the other input parameters pass validation).

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

A resource is made up of one or more subresources, a view identifies which subresources to allow the pipeline to access. In addition, each resource is bound to the pipeline using a view. A shader-resource view is designed to bind any buffer or texture resource to the shader stages using the following API methods: VSSetShaderResources, GSSetShaderResources and PSSetShaderResources.

Since a view is fully typed, this means that typeless resources become fully typed when bound to the pipeline.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::CreateTexture1D method (d3d10.h)

Article • 10/13/20212 minutes to read

Create an array of 1D textures (see Texture1D).

Syntax

C++

```
HRESULT CreateTexture1D(
  [in]  const D3D10_TEXTURE1D_DESC   *pDesc,
  [in]  const D3D10_SUBRESOURCE_DATA *pInitialData,
  [out] ID3D10Texture1D              **ppTexture1D
);
```

Parameters

`[in] pDesc`

Type: **const D3D10_TEXTURE1D_DESC***

Pointer to a 1D texture description (see D3D10_TEXTURE1D_DESC). To create a typeless resource that can be interpreted at runtime into different, compatible formats, specify a typeless format in the texture description. To generate mipmap levels automatically, set the number of mipmap levels to 0.

`[in] pInitialData`

Type: **const D3D10_SUBRESOURCE_DATA***

Pointer to an array of subresource descriptions (see D3D10_SUBRESOURCE_DATA); one for each subresource (ordered by texture array index). Applications may not specify **NULL** for pInitialData when creating IMMUTABLE resources (see D3D10_USAGE). If the resource is multisampled, pInitialData must be **NULL** because multisampled resources cannot be initialized with data when they are created.

`[out] ppTexture1D`

Type: **ID3D10Texture1D****

Address of a pointer to the created texture (see ID3D10Texture1D Interface). Set this parameter to **NULL** to validate the other input parameters (the method will return S_FALSE if the other input parameters pass validation).

Return value

Type: **HRESULT**

If the method succeeds, the return code is S_OK. See Direct3D 10 Return Codes for failing error codes.

Remarks

CreateTexture1D creates a 1D texture resource, which contains an array of 1D textures. The number of textures is specified in the texture description. All textures in a resource must have the same format, size, and number of mipmap levels.

All resources are made up of one or more subresources. To load data into the texture, applications may supply the data initially as part of D3D10_SUBRESOURCE_DATA structure pointed to by pInitialData, or it may use one of the Texturing Functions supplied by the SDK.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::CreateTexture2D method (d3d10.h)

Article • 10/13/20212 minutes to read

Create an array of 2D textures (see Texture2D).

Syntax

C++

```
HRESULT CreateTexture2D(
  [in]  const D3D10_TEXTURE2D_DESC   *pDesc,
  [in]  const D3D10_SUBRESOURCE_DATA *pInitialData,
  [out] ID3D10Texture2D              **ppTexture2D
);
```

Parameters

`[in] pDesc`

Type: **const D3D10_TEXTURE2D_DESC***

Pointer to a 2D texture description (see D3D10_TEXTURE2D_DESC). To create a typeless resource that can be interpreted at runtime into different, compatible formats, specify a typeless format in the texture description. To generate mipmap levels automatically, set the number of mipmap levels to 0.

`[in] pInitialData`

Type: **const D3D10_SUBRESOURCE_DATA***

Pointer to an array of subresource descriptions (see D3D10_SUBRESOURCE_DATA); one for each subresource (ordered by texture array index, then mip level). Applications may not specify **NULL** for pInitialData when creating IMMUTABLE resources (see D3D10_USAGE). If the resource is multisampled, pInitialData must be **NULL** because multisampled resources cannot be initialized with data when they are created.

`[out] ppTexture2D`

Type: **ID3D10Texture2D****

Address of a pointer to the created texture (see ID3D10Texture2D Interface). Set this parameter to **NULL** to validate the other input parameters (the method will return S_FALSE if the other input parameters pass validation).

Return value

Type: **HRESULT**

If the method succeeds, the return code is S_OK. See Direct3D 10 Return Codes for failing error codes.

Remarks

CreateTexture2D creates a 2D texture resource, which contains an array of 1D textures. The number of textures is specified in the texture description. All textures in a resource must have the same format, size, and number of mipmap levels.

All resources are made up of one or more subresources. To load data into the texture, applications may supply the data initially as part of D3D10_SUBRESOURCE_DATA structure pointed to by pInitialData, or it may use one of the Texturing Functions supplied by the SDK.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::CreateTexture3D method (d3d10.h)

Article • 10/13/20212 minutes to read

Create a single 3D texture (see Texture3D).

Syntax

C++

```
HRESULT CreateTexture3D(
  [in]  const D3D10_TEXTURE3D_DESC   *pDesc,
  [in]  const D3D10_SUBRESOURCE_DATA *pInitialData,
  [out] ID3D10Texture3D              **ppTexture3D
);
```

Parameters

`[in] pDesc`

Type: **const D3D10_TEXTURE3D_DESC***

Pointer to a 3D texture description (see D3D10_TEXTURE3D_DESC). To create a typeless resource that can be interpreted at runtime into different, compatible formats, specify a typeless format in the texture description. To generate mipmap levels automatically, set the number of mipmap levels to 0.

`[in] pInitialData`

Type: **const D3D10_SUBRESOURCE_DATA***

Pointer to an array of subresource descriptions (see D3D10_SUBRESOURCE_DATA); one for each subresource (ordered by texture array index, then slice index, then mip level). Applications may not specify **NULL** for pInitialData when creating IMMUTABLE resources (see D3D10_USAGE). If the resource is multisampled, pInitialData must be **NULL** because multisampled resources cannot be initialized with data when they are created.

`[out] ppTexture3D`

Type: **ID3D10Texture3D****

Address of a pointer to the created texture (see ID3D10Texture3D Interface). Set this parameter to **NULL** to validate the other input parameters (the method will return S_FALSE if the other input parameters pass validation).

Return value

Type: **HRESULT**

If the method succeeds, the return code is S_OK. See Direct3D 10 Return Codes for failing error codes.

Remarks

CreateTexture3D creates a 3D texture resource, which contains an array of 1D textures. The number of textures is specified in the texture description. All textures in a resource must have the same format, size, and number of mipmap levels.

All resources are made up of one or more subresources. To load data into the texture, applications may supply the data initially as part of D3D10_SUBRESOURCE_DATA structure pointed to by pInitialData, or it may use one of the Texturing Functions supplied by the SDK.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::CreateVertexShader method (d3d10.h)

Article • 07/27/20222 minutes to read

Create a vertex-shader object from a compiled shader.

Syntax

C++

```
HRESULT CreateVertexShader(
  [in]  const void         *pShaderBytecode,
  [in]  SIZE_T             BytecodeLength,
  [out] ID3D10VertexShader **ppVertexShader
);
```

Parameters

`[in] pShaderBytecode`

Type: **const void***

A pointer to the compiled shader. To get this pointer see Getting a Pointer to a Compiled Shader.

`[in] BytecodeLength`

Type: **SIZE_T**

Size of the compiled vertex shader.

`[out] ppVertexShader`

Type: **ID3D10VertexShader****

Address of a pointer to an ID3D10VertexShader Interface. If this is **NULL**, all other parameters will be validated, and if all parameters pass validation this API will return S_FALSE instead of S_OK.

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::Draw method (d3d10.h)

Article07/27/2022

Draw non-indexed, non-instanced primitives.

Syntax

```
C++

void Draw(
  [in] UINT VertexCount,
  [in] UINT StartVertexLocation
);
```

Parameters

`[in] VertexCount`

Type: **UINT**

Number of vertices to draw.

`[in] StartVertexLocation`

Type: **UINT**

Index of the first vertex, which is usually an offset in a vertex buffer; it could also be used as the first vertex id generated for a shader parameter marked with the **SV_TargetId** system-value semantic.

Return value

None

Remarks

A draw API submits work to the rendering pipeline.

The vertex data for a draw call normally comes from a vertex buffer that is bound to the pipeline. However, you could also provide the vertex data from a shader that has vertex data marked with the **SV_VertexId** system-value semantic.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::DrawAuto method (d3d10.h)

Article • 06/29/20212 minutes to read

Draw geometry of an unknown size that was created by the geometry shader stage. See remarks.

Syntax

```
C++

void DrawAuto();
```

Return value

None

Remarks

A draw API submits work to the rendering pipeline.

After data has been streamed out to SO stage buffers, those buffers can be again bound to the Input Assembler stage at input slot 0 and DrawAuto will draw them without the application needing to know the amount of data that was written to the buffers. A measurement of the amount of data written to the SO stage buffers is maintained internally when the data is streamed out. This means that the CPU does not need to fetch the measurement before re-binding the data that was streamed as input data. Although this amount is tracked internally, it is still the responsibility of applications to use input layouts to describe the format of the data in the SO stage buffers so that the layouts are available when the buffers are again bound to the input assembler.

The following diagram shows the DrawAuto process.

Calling DrawAuto does not change the state of the streaming-output buffers that were bound again as inputs.

DrawAuto only works when drawing with one input buffer bound as an input to the IA stage at slot 0. Applications must create the SO buffer resource with both binding flags, D3D10_BIND_VERTEX_BUFFER and **D3D10_BIND_STREAM_OUTPUT**.

This API does not support indexing or instancing.

If an application needs to retrieve the size of the streaming-output buffer, it can query for statistics on streaming output by using D3D10_QUERY_SO_STATISTICS.

Example of using DrawAuto can be found in the ParticlesGS Sample and PipesGS Sample.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::DrawIndexed method (d3d10.h)

Article • 07/27/20222 minutes to read

Draw indexed, non-instanced primitives.

Syntax

```
C++

void DrawIndexed(
  [in] UINT IndexCount,
  [in] UINT StartIndexLocation,
  [in] INT  BaseVertexLocation
);
```

Parameters

`[in] IndexCount`

Type: **UINT**

Number of indices to draw.

`[in] StartIndexLocation`

Type: **UINT**

Index of the first index to use when accessing the vertex buffer; begin at *StartIndexLocation* to index vertices from the vertex buffer.

`[in] BaseVertexLocation`

Type: **INT**

Offset from the start of the vertex buffer to the first vertex.

Return value

None

Remarks

A draw API submits work to the rendering pipeline.

If the sum of both indices is negative, the result of the function call is undefined.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::DrawIndexedInstanced method (d3d10.h)

Article • 07/27/20222 minutes to read

Draw indexed, instanced primitives.

Syntax

C++

```
void DrawIndexedInstanced(
  [in] UINT IndexCountPerInstance,
  [in] UINT InstanceCount,
  [in] UINT StartIndexLocation,
  [in] INT  BaseVertexLocation,
  [in] UINT StartInstanceLocation
);
```

Parameters

`[in] IndexCountPerInstance`

Type: **UINT**

Size of the index buffer used in each instance.

`[in] InstanceCount`

Type: **UINT**

Number of instances to draw.

`[in] StartIndexLocation`

Type: **UINT**

Index of the first index.

`[in] BaseVertexLocation`

Type: **INT**

Index of the first vertex. The index is signed, which allows a negative index. If the negative index plus the index value from the index buffer are less than 0, the result is undefined.

`[in] StartInstanceLocation`

Type: **UINT**

Index of the first instance.

Return value

None

Remarks

A draw API submits work to the rendering pipeline.

Instancing may extend performance by reusing the same geometry to draw multiple objects in a scene. One example of instancing could be to draw the same object with different positions and colors. Indexing requires multiple vertex buffers: at least one for per-vertex data and a second buffer for per-instance data. For an example of instancing, see the Instancing10 Sample.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::DrawInstanced method (d3d10.h)

Article07/27/2022

Draw non-indexed, instanced primitives.

Syntax

C++

```
void DrawInstanced(
  [in] UINT VertexCountPerInstance,
  [in] UINT InstanceCount,
  [in] UINT StartVertexLocation,
  [in] UINT StartInstanceLocation
);
```

Parameters

`[in] VertexCountPerInstance`

Type: **UINT**

Number of vertices to draw.

`[in] InstanceCount`

Type: **UINT**

Number of instances to draw.

`[in] StartVertexLocation`

Type: **UINT**

Index of the first vertex.

`[in] StartInstanceLocation`

Type: **UINT**

Index of the first instance.

Return value

None

Remarks

A draw API submits work to the rendering pipeline.

Instancing may extend performance by reusing the same geometry to draw multiple objects in a scene. One example of instancing could be to draw the same object with different positions and colors. For an example of instancing, see the Instancing10 Sample.

The vertex data for an instanced draw call normally comes from a vertex buffer that is bound to the pipeline. However, you could also provide the vertex data from a shader that has instanced data identified with a system-value semantic (SV_InstanceID).

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::Flush method (d3d10.h)

Article • 06/29/20212 minutes to read

Send queued-up commands in the command buffer to the GPU.

Syntax

```
C++

void Flush();
```

Return value

None

Remarks

Most applications will not need to call this method. Calling this method when not necessary will incur a performance penalty. Each call to **Flush** incurs a significant amount of overhead.

When Direct3D state-setting, present, or draw commands are called by an application, those commands are queued into an internal command buffer. **Flush** sends those commands to the GPU for processing. Normally, these commands are sent to the GPU automatically whenever Direct3D determines that they need to be, such as when the command buffer is full or when mapping a resource. **Flush** will send the commands manually.

Flush should be used when the CPU waits for an arbitrary amount of time (such as when calling Sleep, ID3DX10ThreadPump::WaitForAllItems, or WaitForVBlank.

For more information about how flushing works, see Accurately Profiling Direct3D API Calls (Direct3D 9).

Requirements

Target Platform	Windows

Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::GSGetConstantBuffers method (d3d10.h)

Article • 07/27/20222 minutes to read

Get the constant buffers used by the geometry shader pipeline stage.

Syntax

```C++
void GSGetConstantBuffers(
  [in]  UINT         StartSlot,
  [in]  UINT         NumBuffers,
  [out] ID3D10Buffer **ppConstantBuffers
);
```

Parameters

`[in] StartSlot`

Type: **UINT**

Index into the device's zero-based array to begin retrieving constant buffers from.

`[in] NumBuffers`

Type: **UINT**

Number of buffers to retrieve.

`[out] ppConstantBuffers`

Type: **ID3D10Buffer****

Array of constant buffer interface pointers (see ID3D10Buffer) to be returned by the method.

Return value

None

Remarks

Any returned interfaces will have their reference count incremented by one. Applications should call IUnknown::Release on the returned interfaces when they are no longer needed to avoid memory leaks.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::GSGetSamplers method (d3d10.h)

Article • 10/13/20212 minutes to read

Get an array of sampler states from the geometry shader pipeline stage.

Syntax

```
C++

void GSGetSamplers(
  [in]  UINT               StartSlot,
  [in]  UINT               NumSamplers,
  [out] ID3D10SamplerState **ppSamplers
);
```

Parameters

`[in] StartSlot`

Type: **UINT**

Index into the device's zero-based array to begin getting samplers from.

`[in] NumSamplers`

Type: **UINT**

Number of samplers to get from the device. Each pipeline stage has a total of 16 sampler slots available.

`[out] ppSamplers`

Type: **ID3D10SamplerState****

Array of sampler-state pointers (see ID3D10SamplerState) to be returned by the device.

Return value

None

Remarks

Any returned interfaces will have their reference count incremented by one. Applications should call IUnknown::Release on the returned interfaces when they are no longer needed to avoid memory leaks.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::GSGetShader method (d3d10.h)

Article • 07/27/20222 minutes to read

Get the geometry shader currently set on the device.

Syntax

```
C++

void GSGetShader(
  [out] ID3D10GeometryShader **ppGeometryShader
);
```

Parameters

`[out] ppGeometryShader`

Type: **ID3D10GeometryShader****

Address of a pointer to a geometry shader (see ID3D10GeometryShader) to be returned by the method.

Return value

None

Remarks

Any returned interfaces will have their reference count incremented by one. Applications should call IUnknown::Release on the returned interfaces when they are no longer needed to avoid memory leaks.

Requirements

Target Platform	Windows

Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::GSGetShaderResources method (d3d10.h)

Article • 07/27/20222 minutes to read

Get the geometry shader resources.

Syntax

```
C++

void GSGetShaderResources(
  [in]  UINT                     StartSlot,
  [in]  UINT                     NumViews,
  [out] ID3D10ShaderResourceView **ppShaderResourceViews
);
```

Parameters

`[in] StartSlot`

Type: **UINT**

Index into the device's zero-based array to begin getting shader resources from.

`[in] NumViews`

Type: **UINT**

The number of resources to get from the device. Up to a maximum of 128 slots are available for shader resources.

`[out] ppShaderResourceViews`

Type: **ID3D10ShaderResourceView****

Array of shader resource view interfaces to be returned by the device.

Return value

None

Remarks

Any returned interfaces will have their reference count incremented by one. Applications should call IUnknown::Release on the returned interfaces when they are no longer needed to avoid memory leaks.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::GSSetConstantBuffers method (d3d10.h)

Article10/13/2021

Set the constant buffers used by the geometry shader pipeline stage.

Syntax

C++

```
void GSSetConstantBuffers(
  [in] UINT         StartSlot,
  [in] UINT         NumBuffers,
  [in] ID3D10Buffer * const *ppConstantBuffers
);
```

Parameters

`[in] StartSlot`

Type: **UINT**

Index into the device's zero-based array to begin setting constant buffers to.

`[in] NumBuffers`

Type: **UINT**

Number of buffers to set.

`[in] ppConstantBuffers`

Type: **ID3D10Buffer***

Array of constant buffers (see ID3D10Buffer) being given to the device.

Return value

None

Remarks

The method will not hold references to the interfaces passed in. For that reason, applications should be careful not to release interfaces currently in use by the device.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::GSSetSamplers method (d3d10.h)

Article • 07/27/20222 minutes to read

Set an array of sampler states to the geometry shader pipeline stage.

Syntax

```C++
void GSSetSamplers(
  [in] UINT                StartSlot,
  [in] UINT                NumSamplers,
  [in] ID3D10SamplerState * const *ppSamplers
);
```

Parameters

`[in] StartSlot`

Type: **UINT**

Index into the device's zero-based array to begin setting samplers to.

`[in] NumSamplers`

Type: **UINT**

Number of samplers in the array. Each pipeline stage has a total of 16 sampler slots available.

`[in] ppSamplers`

Type: **ID3D10SamplerState***

Pointer to an array of sampler-state interfaces (see ID3D10SamplerState). See Remarks.

Return value

None

Remarks

Any sampler may be set to **NULL**; this invokes the default state, which is defined to be the following.

```
//Default sampler state:
D3D10_SAMPLER_DESC SamplerDesc;
SamplerDesc.Filter = D3D10_FILTER_MIN_MAG_MIP_LINEAR;
SamplerDesc.AddressU = D3D10_TEXTURE_ADDRESS_CLAMP;
SamplerDesc.AddressV = D3D10_TEXTURE_ADDRESS_CLAMP;
SamplerDesc.AddressW = D3D10_TEXTURE_ADDRESS_CLAMP;
SamplerDesc.MipLODBias = 0;
SamplerDesc.MaxAnisotropy = 1;
SamplerDesc.ComparisonFunc = D3D10_COMPARISON_NEVER;
SamplerDesc.BorderColor[0] = 1.0f;
SamplerDesc.BorderColor[1] = 1.0f;
SamplerDesc.BorderColor[2] = 1.0f;
SamplerDesc.BorderColor[3] = 1.0f;
SamplerDesc.MinLOD = -FLT_MAX;
SamplerDesc.MaxLOD = FLT_MAX;
```

The method will not hold a reference to the interfaces passed in. For that reason, applications should be careful not to release an interface currently in use by the device.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::GSSetShader method (d3d10.h)

Article • 07/27/20222 minutes to read

Set a geometry shader to the device.

Syntax

```
C++

void GSSetShader(
  [in] ID3D10GeometryShader *pShader
);
```

Parameters

`[in] pShader`

Type: **ID3D10GeometryShader***

Pointer to a geometry shader (see ID3D10GeometryShader). Passing in **NULL** disables the shader for this pipeline stage.

Return value

None

Remarks

The method will not hold a reference to the interfaces passed in. For that reason, applications should be careful not to release an interface currently in use by the device.

Requirements

Target Platform	Windows

Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::GSSetShaderResources method (d3d10.h)

Article • 07/27/20222 minutes to read

Bind an array of shader resources to the geometry shader stage.

Syntax

C++

```
void GSSetShaderResources(
  [in] UINT                     StartSlot,
  [in] UINT                     NumViews,
  [in] ID3D10ShaderResourceView * const *ppShaderResourceViews
);
```

Parameters

`[in] StartSlot`

Type: **UINT**

Index into the device's zero-based array to begin setting shader resources to.

`[in] NumViews`

Type: **UINT**

Number of shader resources to set. Up to a maximum of 128 slots are available for shader resources.

`[in] ppShaderResourceViews`

Type: **ID3D10ShaderResourceView***

Array of shader resource view interfaces to set to the device.

Return value

None

Remarks

If you bind a subresource as an input and an output, this API will fill the destination shader resource slot with **NULL**. The debug layer (when active) will alert you if this is true.

For information about creating shader-resource views, see ID3D10Device::CreateShaderResourceView.

The method will not hold a reference to the interfaces passed in. For that reason, applications should be careful not to release an interface currently in use by the device.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::GenerateMips method (d3d10.h)

Article • 07/27/20222 minutes to read

Generates mipmaps for the given shader resource.

Syntax

```
C++

void GenerateMips(
  [in] ID3D10ShaderResourceView *pShaderResourceView
);
```

Parameters

`[in] pShaderResourceView`

Type: **ID3D10ShaderResourceView***

A pointer to an ID3D10ShaderResourceView. The mipmaps will be generated for this shader resource.

Return value

None

Remarks

GenerateMips may be called on any ID3D10ShaderResourceView in order to generate the lower mipmap levels. GenerateMips uses the largest mipmap level of the view to recursively generate the lower levels of the mip, stopping with the smallest level specified by the view. If the base resource was not created with D3D10_BIND_RENDER_TARGET and D3D10_RESOURCE_MISC_GENERATE_MIPS, this call has no effect.

Video adapters that support feature level 9.1 and higher support generating mipmaps if you use any of these formats:

```
DXGI_FORMAT_R8G8B8A8_UNORM
DXGI_FORMAT_R8G8B8A8_UNORM_SRGB
DXGI_FORMAT_B5G6R5_UNORM
DXGI_FORMAT_B8G8R8A8_UNORM
DXGI_FORMAT_B8G8R8A8_UNORM_SRGB
DXGI_FORMAT_B8G8R8X8_UNORM
DXGI_FORMAT_B8G8R8X8_UNORM_SRGB
```

Video adapters that support feature level 9.2 and higher support generating mipmaps if you use any of these formats in addition to any of the formats for feature level 9.1:

```
DXGI_FORMAT_R16G16B16A16_FLOAT
DXGI_FORMAT_R16G16B16A16_UNORM
DXGI_FORMAT_R16G16_FLOAT
DXGI_FORMAT_R16G16_UNORM
DXGI_FORMAT_R32_FLOAT
```

Video adapters that support feature level 9.3 and higher support generating mipmaps if you use any of these formats in addition to any of the formats for feature levels 9.1 and 9.2:

```
DXGI_FORMAT_R32G32B32A32_FLOAT
DXGI_FORMAT_B4G4R4A4 (optional)
```

Video adapters that support feature level 10 and higher support generating mipmaps if you use any of these formats in addition to any of the formats for feature levels 9.1, 9.2, and 9.3:

```
DXGI_FORMAT_R32G32B32_FLOAT (optional)
DXGI_FORMAT_R16G16B16A16_SNORM
DXGI_FORMAT_R32G32_FLOAT
DXGI_FORMAT_R10G10B10A2_UNORM
DXGI_FORMAT_R11G11B10_FLOAT
```

```
DXGI_FORMAT_R8G8B8A8_SNORM
DXGI_FORMAT_R16G16_SNORM
DXGI_FORMAT_R8G8_UNORM
DXGI_FORMAT_R8G8_SNORM
DXGI_FORMAT_R16_FLOAT
DXGI_FORMAT_R16_UNORM
DXGI_FORMAT_R16_SNORM
DXGI_FORMAT_R8_UNORM
DXGI_FORMAT_R8_SNORM
DXGI_FORMAT_A8_UNORM
DXGI_FORMAT_B5G5R5A1_UNORM (optional)
```

For all other unsupported formats, this method will silently fail.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

DXGI_FORMAT

ID3D10Device Interface

ID3D10Device::GetCreationFlags method (d3d10.h)

Article • 06/29/20212 minutes to read

Get the flags used during the call to create the device with D3D10CreateDevice.

Syntax

```
C++

UINT GetCreationFlags();
```

Return value

Type: **UINT**

A bitfield containing the flags used to create the device. See D3D10_CREATE_DEVICE_FLAG.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::GetDeviceRemovedReason method (d3d10.h)

Article • 07/27/20222 minutes to read

Get the reason why the device was removed.

Syntax

```
C++

HRESULT GetDeviceRemovedReason();
```

Return value

Type: **HRESULT**

Possible return values include:

- DXGI_ERROR_DEVICE_HUNG
- DXGI_ERROR_DEVICE_REMOVED
- DXGI_ERROR_DEVICE_RESET
- DXGI_ERROR_DRIVER_INTERNAL_ERROR
- DXGI_ERROR_INVALID_CALL
- S_OK

For more detail on these return codes, see DXGI_ERROR.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::GetExceptionMode method (d3d10.h)

Article • 07/27/20222 minutes to read

Get the exception-mode flags.

Syntax

```
C++

UINT GetExceptionMode();
```

Return value

Type: **UINT**

A value that contains one or more exception flags; each flag specifies a condition which will cause an exception to be raised. The flags are listed in D3D10_RAISE_FLAG. A default value of 0 means there are no flags.

Remarks

An exception-mode flag is used to elevate an error condition to a non-continuable exception.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::GetPredication method (d3d10.h)

Article • 07/27/20222 minutes to read

Get the rendering predicate state.

Syntax

```
C++

void GetPredication(
  [out] ID3D10Predicate **ppPredicate,
  [out] BOOL            *pPredicateValue
);
```

Parameters

`[out] ppPredicate`

Type: **ID3D10Predicate****

Address of a pointer to a predicate (see ID3D10Predicate). Value stored here will be **NULL** upon device creation.

`[out] pPredicateValue`

Type: **BOOL***

Address of a boolean to fill with the predicate comparison value. **FALSE** upon device creation.

Return value

None

Remarks

Any returned interfaces will have their reference count incremented by one. Applications should call IUnknown::Release on the returned interfaces when they are no longer needed to avoid memory leaks.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::GetPrivateData method (d3d10.h)

Article • 10/13/20212 minutes to read

Get data from a device that is associated with a guid.

Syntax

C++

```
HRESULT GetPrivateData(
  [in]      REFGUID guid,
  [in, out] UINT    *pDataSize,
  [out]     void    *pData
);
```

Parameters

`[in] guid`

Type: **REFGUID**

Guid associated with the data.

`[in, out] pDataSize`

Type: **UINT***

Size of the data.

`[out] pData`

Type: **void***

Pointer to the data stored with the device. If pData is **NULL**, DataSize must also be 0, and any data previously associated with the guid will be destroyed.

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

The data stored in the device is set with ID3D10Device::SetPrivateData.

The data retrieved and the guid will typically be application-defined.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::GetTextFilterSize method (d3d10.h)

Article07/27/2022

This method is not implemented.

Syntax

```
C++

void GetTextFilterSize(
  UINT *pWidth,
  UINT *pHeight
);
```

Parameters

`pWidth`

Type: **UINT***

Not applicable

`pHeight`

Type: **UINT***

Not applicable

Return value

None

Remarks

This method is not implemented, and should not be used.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::IAGetIndexBuffer method (d3d10.h)

Article07/27/2022

Get a pointer to the index buffer that is bound to the input-assembler stage.

Syntax

C++

```
void IAGetIndexBuffer(
  [out] ID3D10Buffer **pIndexBuffer,
  [out] DXGI_FORMAT  *Format,
  [out] UINT         *Offset
);
```

Parameters

`[out] pIndexBuffer`

Type: **ID3D10Buffer****

A pointer to an index buffer returned by the method (see ID3D10Buffer).

`[out] Format`

Type: **DXGI_FORMAT***

Specifies format of the data in the index buffer (see DXGI_FORMAT). These formats provide the size and type of the data in the buffer. The only formats allowed for index buffer data are 16-bit (DXGI_FORMAT_R16_UINT) and 32-bit (DXGI_FORMAT_R32_UINT) integers.

`[out] Offset`

Type: **UINT***

Offset (in bytes) from the start of the index buffer, to the first index to use.

Return value

None

Remarks

Any returned interfaces will have their reference count incremented by one. Applications should call IUnknown::Release on the returned interfaces when they are no longer needed to avoid memory leaks.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::IAGetInputLayout method (d3d10.h)

Article • 07/27/20222 minutes to read

Get a pointer to the input-layout object that is bound to the input-assembler stage.

Syntax

C++

```
void IAGetInputLayout(
  [out] ID3D10InputLayout **ppInputLayout
);
```

Parameters

`[out] ppInputLayout`

Type: **ID3D10InputLayout****

A pointer to the input-layout object (see ID3D10InputLayout), which describes the input buffers that will be read by the IA stage.

Return value

None

Remarks

For information about creating an input-layout object, see Creating the Input-Layout Object.

Any returned interfaces will have their reference count incremented by one. Applications should call IUnknown::Release on the returned interfaces when they are no longer needed to avoid memory leaks.

Requirements

Target Platform	Windows
Header	d3d10.h

See also

ID3D10Device Interface

ID3D10Device::IAGetPrimitiveTopology method (d3d10.h)

Article • 07/27/20222 minutes to read

Get information about the primitive type, and data order that describes input data for the input assembler stage.

Syntax

```
C++

void IAGetPrimitiveTopology(
  [out] D3D10_PRIMITIVE_TOPOLOGY *pTopology
);
```

Parameters

`[out] pTopology`

Type: **D3D10_PRIMITIVE_TOPOLOGY***

A pointer to the type of primitive, and ordering of the primitive data (see D3D10_PRIMITIVE_TOPOLOGY).

Return value

None

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::IAGetVertexBuffers method (d3d10.h)

Article • 07/27/20222 minutes to read

Get the vertex buffers bound to the input-assembler stage.

Syntax

C++

```
void IAGetVertexBuffers(
  [in]  UINT         StartSlot,
  [in]  UINT         NumBuffers,
  [out] ID3D10Buffer **ppVertexBuffers,
  [out] UINT         *pStrides,
  [out] UINT         *pOffsets
);
```

Parameters

`[in] StartSlot`

Type: **UINT**

The input slot of the first vertex buffer to get. The first vertex buffer is explicitly bound to the start slot; this causes each additional vertex buffer in the array to be implicitly bound to each subsequent input slot. A maximum of 16 or 32 input slots (ranges from 0 to either D3D10_IA_VERTEX_INPUT_RESOURCE_SLOT_COUNT - 1 or D3D10_1_IA_VERTEX_INPUT_RESOURCE_SLOT_COUNT - 1) are available; the maximum number of input slots depends on the feature level.

`[in] NumBuffers`

Type: **UINT**

The number of vertex buffers to get starting at the offset. The number of buffers (plus the starting slot) cannot exceed the total number of IA-stage input slots.

`[out] ppVertexBuffers`

Type: **ID3D10Buffer****

A pointer to an array of vertex buffers returned by the method (see ID3D10Buffer).

`[out] pStrides`

Type: **UINT***

Pointer to an array of stride values returned by the method; one stride value for each buffer in the vertex-buffer array. Each stride value is the size (in bytes) of the elements that are to be used from that vertex buffer.

`[out] pOffsets`

Type: **UINT***

Pointer to an array of offset values returned by the method; one offset value for each buffer in the vertex-buffer array. Each offset is the number of bytes between the first element of a vertex buffer and the first element that will be used.

Return value

None

Remarks

Any returned interfaces will have their reference count incremented by one. Applications should call IUnknown::Release on the returned interfaces when they are no longer needed to avoid memory leaks.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::IASetIndexBuffer method (d3d10.h)

Article • 07/27/20222 minutes to read

Bind an index buffer to the input-assembler stage.

Syntax

C++

```
void IASetIndexBuffer(
  [in] ID3D10Buffer *pIndexBuffer,
  [in] DXGI_FORMAT  Format,
  [in] UINT         Offset
);
```

Parameters

`[in] pIndexBuffer`

Type: **ID3D10Buffer***

A pointer to a buffer (see ID3D10Buffer) that contains indices. The index buffer must have been created with the D3D10_BIND_INDEX_BUFFER flag.

`[in] Format`

Type: **DXGI_FORMAT**

Specifies format of the data in the index buffer. The only formats allowed for index buffer data are 16-bit (DXGI_FORMAT_R16_UINT) and 32-bit (**DXGI_FORMAT_R32_UINT**) integers.

`[in] Offset`

Type: **UINT**

Offset (in bytes) from the start of the index buffer to the first index to use.

Return value

None

Remarks

For information about creating index buffers, see Create an Index Buffer.

Calling this method using a buffer that is currently bound for writing (i.e. bound to the stream output pipeline stage) will effectively bind **NULL** instead because a buffer cannot be bound as both an input and an output at the same time.

The Debug Layer will generate a warning whenever a resource is prevented from being bound simultaneously as an input and an output, but this will not prevent invalid data from being used by the runtime.

The method will not hold a reference to the interfaces passed in. For that reason, applications should be careful not to release an interface currently in use by the device.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::IASetInputLayout method (d3d10.h)

Article • 07/27/20222 minutes to read

Bind an input-layout object to the input-assembler stage.

Syntax

```
C++

void IASetInputLayout(
  [in] ID3D10InputLayout *pInputLayout
);
```

Parameters

`[in] pInputLayout`

Type: **ID3D10InputLayout***

A pointer to the input-layout object (see ID3D10InputLayout), which describes the input buffers that will be read by the IA stage.

Return value

None

Remarks

Input-layout objects describe how vertex buffer data is streamed into the IA pipeline stage. To create an input-layout object, call ID3D10Device::CreateInputLayout.

The method will not hold a reference to the interfaces passed in. For that reason, applications should be careful not to release an interface currently in use by the device.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::IASetPrimitiveTopology method (d3d10.h)

Article • 07/27/20222 minutes to read

Bind information about the primitive type, and data order that describes input data for the input assembler stage.

Syntax

```
C++

void IASetPrimitiveTopology(
  [in] D3D10_PRIMITIVE_TOPOLOGY Topology
);
```

Parameters

`[in] Topology`

Type: **D3D10_PRIMITIVE_TOPOLOGY**

The type of primitive and ordering of the primitive data (see D3D10_PRIMITIVE_TOPOLOGY).

Return value

None

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::IASetVertexBuffers method (d3d10.h)

Article07/27/2022

Bind an array of vertex buffers to the input-assembler stage.

Syntax

C++

```
void IASetVertexBuffers(
  [in] UINT         StartSlot,
  [in] UINT         NumBuffers,
  [in] ID3D10Buffer * const *ppVertexBuffers,
  [in] const UINT   *pStrides,
  [in] const UINT   *pOffsets
);
```

Parameters

`[in] StartSlot`

Type: **UINT**

The first input slot for binding. The first vertex buffer is explicitly bound to the start slot; this causes each additional vertex buffer in the array to be implicitly bound to each subsequent input slot. A maximum of 16 or 32 input slots (ranges from 0 to either D3D10_IA_VERTEX_INPUT_RESOURCE_SLOT_COUNT - 1 or D3D10_1_IA_VERTEX_INPUT_RESOURCE_SLOT_COUNT - 1) are available; the maximum number of input slots depends on the feature level.

`[in] NumBuffers`

Type: **UINT**

The number of vertex buffers in the array. The number of buffers (plus the starting slot) cannot exceed the total number of IA-stage input slots.

`[in] ppVertexBuffers`

Type: **ID3D10Buffer***

A pointer to an array of vertex buffers (see ID3D10Buffer). The vertex buffers must have been created with the D3D10_BIND_VERTEX_BUFFER flag.

`[in] pStrides`

Type: **const UINT***

Pointer to an array of stride values; one stride value for each buffer in the vertex-buffer array. Each stride is the size (in bytes) of the elements that are to be used from that vertex buffer.

`[in] pOffsets`

Type: **const UINT***

Pointer to an array of offset values; one offset value for each buffer in the vertex-buffer array. Each offset is the number of bytes between the first element of a vertex buffer and the first element that will be used.

Return value

None

Remarks

For information about creating vertex buffers, see Create a Vertex Buffer.

Calling this method using a buffer that is currently bound for writing (i.e. bound to the stream output pipeline stage) will effectively bind **NULL** instead because a buffer cannot be bound as both an input and an output at the same time.

The Debug Layer will generate a warning whenever a resource is prevented from being bound simultaneously as an input and an output, but this will not prevent invalid data from being used by the runtime.

The method will not hold a reference to the interfaces passed in. For that reason, applications should be careful not to release an interface currently in use by the device.

Requirements

Target Platform	Windows

Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::OMGetBlendState method (d3d10.h)

Article • 07/27/20222 minutes to read

Get the blend state of the output-merger stage.

Syntax

```
C++

void OMGetBlendState(
  [out] ID3D10BlendState **ppBlendState,
  [out] FLOAT [4]        BlendFactor,
  [out] UINT             *pSampleMask
);
```

Parameters

`[out] ppBlendState`

Type: **ID3D10BlendState****

Address of a pointer to a blend-state interface (see ID3D10BlendState).

`[out] BlendFactor`

Type: **FLOAT**

Array of blend factors, one for each RGBA component.

`[out] pSampleMask`

Type: **UINT***

Pointer to a sample mask.

Return value

None

Remarks

The reference count of the returned interface will be incremented by one when the blend state is retrieved. Applications must release returned pointer(s) when they are no longer needed, or else there will be a memory leak.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::OMGetDepthStencilState method (d3d10.h)

Article07/27/2022

Gets the depth-stencil state of the output-merger stage.

Syntax

C++

```
void OMGetDepthStencilState(
  [out] ID3D10DepthStencilState **ppDepthStencilState,
  [out] UINT                    *pStencilRef
);
```

Parameters

`[out] ppDepthStencilState`

Type: **ID3D10DepthStencilState****

Address of a pointer to a depth-stencil state interface (see ID3D10DepthStencilState) to be filled with information from the device.

`[out] pStencilRef`

Type: **UINT***

Pointer to the stencil reference value used in the depth-stencil test.

Return value

None

Remarks

Any returned interfaces will have their reference count incremented by one. Applications should call IUnknown::Release on the returned interfaces when they are no longer needed to avoid memory leaks.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::OMGetRenderTargets method (d3d10.h)

Article10/13/2021

Get pointers to the render targets and the depth-stencil buffer that are available to the output-merger stage.

Syntax

```C++
void OMGetRenderTargets(
  [in]  UINT                    NumViews,
  [out] ID3D10RenderTargetView **ppRenderTargetViews,
  [out] ID3D10DepthStencilView **ppDepthStencilView
);
```

Parameters

`[in] NumViews`

Type: **UINT**

Number of render targets to retrieve.

`[out] ppRenderTargetViews`

Type: **ID3D10RenderTargetView****

Pointer to an array of render targets views (see ID3D10RenderTargetView) to be filled with the render targets from the device. Specify **NULL** for this parameter when retrieval of a render target is not needed.

`[out] ppDepthStencilView`

Type: **ID3D10DepthStencilView****

Pointer to a depth-stencil view (see ID3D10DepthStencilView) to be filled with the depth-stencil information from the device. Specify **NULL** for this parameter when retrieval of the depth-stencil view is not needed.

Return value

None

Remarks

Any returned interfaces will have their reference count incremented by one. Applications should call IUnknown::Release on the returned interfaces when they are no longer needed to avoid memory leaks.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::OMSetBlendState method (d3d10.h)

Article • 07/27/20222 minutes to read

Set the blend state of the output-merger stage.

Syntax

```
C++

void OMSetBlendState(
  [in] ID3D10BlendState *pBlendState,
  [in] const FLOAT [4]  BlendFactor,
  [in] UINT             SampleMask
);
```

Parameters

`[in] pBlendState`

Type: **ID3D10BlendState***

Pointer to a blend-state interface (see ID3D10BlendState). Passing in **NULL** implies a default blend state. See remarks for further details.

`[in] BlendFactor`

Type: **const FLOAT**

Array of blend factors, one for each RGBA component. The blend factors modulate values for the pixel shader, render target, or both. If you created the blend-state object with D3D10_BLEND_BLEND_FACTOR or D3D10_BLEND_INV_BLEND_FACTOR, the blending stage uses the non-NULL array of blend factors. If you didn't create the blend-state object with **D3D10_BLEND_BLEND_FACTOR** or **D3D10_BLEND_INV_BLEND_FACTOR**, the blending stage does not use the non-NULL array of blend factors; the runtime stores the blend factors, and you can later call ID3D11DeviceContext::OMGetBlendState to retrieve the blend factors. If you pass **NULL**, the runtime uses or stores a blend factor equal to { 1, 1, 1, 1 }.

`[in] SampleMask`

Type: **UINT**

32-bit sample coverage. The default value is 0xffffffff. See remarks.

Return value

None

Remarks

Blend state is used by the output-merger stage to determine how to blend together two RGB pixel values and two alpha values. The two RGB pixel values and two alpha values are the RGB pixel value and alpha value that the pixel shader outputs and the RGB pixel value and alpha value already in the output render target. The blend option controls the data source that the blending stage uses to modulate values for the pixel shader, render target, or both. The blend operation controls how the blending stage mathematically combines these modulated values.

To create a blend-state interface, call ID3D10Device::CreateBlendState.

Passing in **NULL** for the blend-state interface indicates to the runtime to set a default blending state. The following table indicates the default blending parameters.

State	Default Value
AlphaToCoverageEnable	**FALSE**
BlendEnable	**FALSE**[8]
SrcBlend	D3D10_BLEND_ONE
DstBlend	D3D10_BLEND_ZERO
BlendOp	D3D10_BLEND_OP_ADD
SrcBlendAlpha	D3D10_BLEND_ONE
DstBlendAlpha	D3D10_BLEND_ZERO
BlendOpAlpha	D3D10_BLEND_OP_ADD
RenderTargetWriteMask[8]	D3D10_COLOR_WRITE_ENABLE_ALL[8]

A sample mask determines which samples get updated in all the active render targets. The mapping of bits in a sample mask to samples in a multisample render target is the

responsibility of an individual application. A sample mask is always applied; it is independent of whether multisampling is enabled, and does not depend on whether an application uses multisample render targets.

The method will not hold a reference to the interfaces passed in. For that reason, applications should be careful not to release an interface currently in use by the device.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::OMSetDepthStencilState method (d3d10.h)

Article • 07/27/20222 minutes to read

Sets the depth-stencil state of the output-merger stage.

Syntax

C++

```
void OMSetDepthStencilState(
  [in] ID3D10DepthStencilState *pDepthStencilState,
  [in] UINT                    StencilRef
);
```

Parameters

`[in] pDepthStencilState`

Type: **ID3D10DepthStencilState***

Pointer to a depth-stencil state interface (see ID3D10DepthStencilState) to bind to the device.

`[in] StencilRef`

Type: **UINT**

Reference value to perform against when doing a depth-stencil test. See remarks.

Return value

None

Remarks

To create a depth-stencil state interface, call ID3D10Device::CreateDepthStencilState.

Depth-stencil state is used by the output-merger stage to setup depth-stencil testing. The stencil reference value is the control value used in the depth-stencil test.

The method will not hold a reference to the interfaces passed in. For that reason, applications should be careful not to release an interface currently in use by the device.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::OMSetRenderTargets method (d3d10.h)

Article • 10/13/20212 minutes to read

Bind one or more render targets and the depth-stencil buffer to the output-merger stage.

Syntax

C++

```
void OMSetRenderTargets(
  [in] UINT                    NumViews,
  [in] ID3D10RenderTargetView * const *ppRenderTargetViews,
  [in] ID3D10DepthStencilView *pDepthStencilView
);
```

Parameters

`[in] NumViews`

Type: **UINT**

Number of render targets to bind.

`[in] ppRenderTargetViews`

Type: **ID3D10RenderTargetView***

Pointer to an array of render targets (see ID3D10RenderTargetView) to bind to the device. If this parameter is **NULL**, no render targets are bound. See Remarks.

`[in] pDepthStencilView`

Type: **ID3D10DepthStencilView***

Pointer to a depth-stencil view (see ID3D10DepthStencilView) to bind to the device. If this parameter is **NULL**, the depth-stencil state is not bound.

Return value

None

Remarks

A call to **OMSetRenderTargets** overrides all bounded render targets and the depth stencil target regardless of the number of render targets in *ppRenderTargetViews*.

The maximum number of render targets a device can have active at any given time is set by a #define in D3D10.h called D3D10_SIMULTANEOUS_RENDER_TARGET_COUNT. It is invalid to try to set the same subresource to multiple render target slots.

If any subresources are also currently bound for reading or writing (perhaps in a different part of the pipeline), those bind points will be **NULL**'ed out to prevent the same subresource from being read and written simultaneously in a single rendering operation.

The method will not hold references to the interfaces passed in. For that reason, applications should be careful not to release interfaces currently in use by the device.

See Binding Resources and Pipeline stages for more information on binding resources.

The pixel shader must be able to simultaneously render to at least eight separate render targets. All of these render targets must access the same type of resource: Buffer, Texture1D, Texture1DArray, Texture2D, Texture2DArray, Texture3D, or TextureCube. All render targets must have the same size in all dimensions (width and height, and depth for 3D or array size for *Array types). If render targets use multisample anti-aliasing, all bound render targets and depth buffer must be the same form of multisample resource (that is, the sample counts must be the same). Each render target can have a different data format. These render target formats are not required to have identical bit-per-element counts.

Any combination of the eight slots for render targets can have a render target set or not set.

The same resource view cannot be bound to multiple render target slots simultaneously. However, you can set multiple non-overlapping resource views of a single resource as simultaneous multiple render targets.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::OpenSharedResource method (d3d10.h)

Article • 10/13/20212 minutes to read

Give a device access to a shared resource created on a different Direct3d device.

Syntax

C++

```
HRESULT OpenSharedResource(
  [in]  HANDLE hResource,
  [in]  REFIID ReturnedInterface,
  [out] void   **ppResource
);
```

Parameters

`[in] hResource`

Type: **HANDLE**

A resource handle. See remarks.

`[in] ReturnedInterface`

Type: **REFIID**

The globally unique identifier (GUID) for the resource interface. See remarks.

`[out] ppResource`

Type: **void****

Address of a pointer to the resource we are gaining access to.

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

To share a resource between two Direct3D 10 devices the resource must have been created with the D3D10_RESOURCE_MISC_SHARED flag, if it was created using the ID3D10Device interface. If it was created using the IDXGIDevice interface, then the resource is always shared.

The REFIID, or GUID, of the interface to the resource can be obtained by using the __uuidof() macro. For example, __uuidof(ID3D10Buffer) will get the GUID of the interface to a buffer resource.

When sharing a resource between two Direct3D 10 devices the unique handle of the resource can be obtained by querying the resource for the IDXGIResource interface and then calling GetSharedHandle.

```
IDXGIResource* pOtherResource(NULL);
hr = pOtherDeviceResource->QueryInterface( __uuidof(IDXGIResource),
(void**)&pOtherResource );
HANDLE sharedHandle;
pOtherResource->GetSharedHandle(&sharedHandle);
```

The only resources that can be shared are 2D non-mipmapped textures.

To share a resource between a Direct3D 9 device and a Direct3D 10 device the texture must have been created using the *pSharedHandle* argument of CreateTexture.
The shared Direct3D 9 handle is then passed to OpenSharedResource in the *hResource* argument.

The following code illustrates the method calls involved.

```
sharedHandle = NULL; // must be set to NULL to create, can use a valid
handle here to open in D3D9
pDevice9->CreateTexture(..., pTex2D_9, &sharedHandle);
...
pDevice10->OpenSharedResource(sharedHandle, __uuidof(ID3D10Resource),
(void**)(&tempResource10));
tempResource10->QueryInterface(__uuidof(ID3D10Texture2D), (void**)
(&pTex2D_10));
tempResource10->Release();
```

```
// now use pTex2D_10 with pDevice10
```

Textures being shared from D3D9 to D3D10 have the following restrictions.

- Textures must be 2D
- Only 1 mip level is allowed
- Texture must have default usage
- Texture must be write only
- MSAA textures are not allowed
- Bind flags must have SHADER_RESOURCE and RENDER_TARGET set
- Only R10G10B10A2_UNORM, R16G16B16A16_FLOAT and R8G8B8A8_UNORM formats are allowed

If a shared texture is updated on one device ID3D10Device::Flush must be called on that device.

Requirements

Target Platform	Windows
Header	d3d10.h

See also

ID3D10Device Interface

ID3D10Device::PSGetConstantBuffers method (d3d10.h)

Article07/27/2022

Get the constant buffers used by the pixel shader pipeline stage.

Syntax

```
C++

void PSGetConstantBuffers(
  [in]  UINT         StartSlot,
  [in]  UINT         NumBuffers,
  [out] ID3D10Buffer **ppConstantBuffers
);
```

Parameters

`[in] StartSlot`

Type: **UINT**

Index into the device's zero-based array to begin retrieving constant buffers from.

`[in] NumBuffers`

Type: **UINT**

Number of buffers to retrieve.

`[out] ppConstantBuffers`

Type: **ID3D10Buffer****

Array of constant buffer interface pointers (see ID3D10Buffer) to be returned by the method.

Return value

None

Remarks

Any returned interfaces will have their reference count incremented by one. Applications should call IUnknown::Release on the returned interfaces when they are no longer needed to avoid memory leaks.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::PSGetSamplers method (d3d10.h)

Article07/27/2022

Get an array of sampler states from the pixel shader pipeline stage.

Syntax

```
C++

void PSGetSamplers(
  [in]  UINT               StartSlot,
  [in]  UINT               NumSamplers,
  [out] ID3D10SamplerState **ppSamplers
);
```

Parameters

`[in] StartSlot`

Type: **UINT**

Index into the device's zero-based array to begin getting samplers from.

`[in] NumSamplers`

Type: **UINT**

Number of samplers to get from the device. Each pipeline stage has a total of 16 sampler slots available.

`[out] ppSamplers`

Type: **ID3D10SamplerState****

Array of sampler-state interface pointers (see ID3D10SamplerState) to be returned by the device.

Return value

None

Remarks

Any returned interfaces will have their reference count incremented by one. Applications should call IUnknown::Release on the returned interfaces when they are no longer needed to avoid memory leaks.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::PSGetShader method (d3d10.h)

Article • 07/27/20222 minutes to read

Get the pixel shader currently set on the device.

Syntax

```
C++

void PSGetShader(
  [out] ID3D10PixelShader **ppPixelShader
);
```

Parameters

`[out] ppPixelShader`

Type: **ID3D10PixelShader****

Address of a pointer to a pixel shader (see ID3D10PixelShader) to be returned by the method.

Return value

None

Remarks

Any returned interfaces will have their reference count incremented by one. Applications should call IUnknown::Release on the returned interfaces when they are no longer needed to avoid memory leaks.

Requirements

Target Platform	Windows

Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::PSGetShaderResources method (d3d10.h)

Article • 07/27/20222 minutes to read

Get the pixel shader resources.

Syntax

```
C++

void PSGetShaderResources(
  [in]  UINT                     StartSlot,
  [in]  UINT                     NumViews,
  [out] ID3D10ShaderResourceView **ppShaderResourceViews
);
```

Parameters

`[in] StartSlot`

Type: **UINT**

Index into the device's zero-based array to begin getting shader resources from.

`[in] NumViews`

Type: **UINT**

The number of resources to get from the device. Up to a maximum of 128 slots are available for shader resources.

`[out] ppShaderResourceViews`

Type: **ID3D10ShaderResourceView****

Array of shader resource view interfaces to be returned by the device.

Return value

None

Remarks

Any returned interfaces will have their reference count incremented by one. Applications should call IUnknown::Release on the returned interfaces when they are no longer needed to avoid memory leaks.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::PSSetConstantBuffers method (d3d10.h)

Article • 10/13/20212 minutes to read

Set the constant buffers used by the pixel shader pipeline stage.

Syntax

C++

```
void PSSetConstantBuffers(
  [in] UINT         StartSlot,
  [in] UINT         NumBuffers,
  [in] ID3D10Buffer * const *ppConstantBuffers
);
```

Parameters

`[in] StartSlot`

Type: **UINT**

Index into the device's zero-based array to begin setting constant buffers to.

`[in] NumBuffers`

Type: **UINT**

Number of buffers to set.

`[in] ppConstantBuffers`

Type: **ID3D10Buffer***

Array of constant buffers (see ID3D10Buffer) being given to the device.

Return value

None

Remarks

The method will not hold a reference to the interfaces passed in. For that reason, applications should be careful not to release an interface currently in use by the device.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::PSSetSamplers method (d3d10.h)

Article • 07/27/20222 minutes to read

Set an array of sampler states to the pixel shader pipeline stage.

Syntax

```C++
void PSSetSamplers(
  [in] UINT               StartSlot,
  [in] UINT               NumSamplers,
  [in] ID3D10SamplerState * const *ppSamplers
);
```

Parameters

`[in] StartSlot`

Type: **UINT**

Index into the device's zero-based array to begin setting samplers to.

`[in] NumSamplers`

Type: **UINT**

Number of samplers in the array. Each pipeline stage has a total of 16 sampler slots available.

`[in] ppSamplers`

Type: **ID3D10SamplerState***

Pointer to an array of sampler-state interfaces (see ID3D10SamplerState). See Remarks.

Return value

None

Remarks

Any sampler may be set to **NULL**; this invokes the default state, which is defined to be the following.

State	Default Value
Filter	D3D10_FILTER_MIN_MAG_MIP_LINEAR
AddressU	D3D10_TEXTURE_ADDRESS_CLAMP
AddressV	D3D10_TEXTURE_ADDRESS_CLAMP
AddressW	D3D10_TEXTURE_ADDRESS_CLAMP
MipLODBias	0
MaxAnisotropy	1
ComparisonFunc	D3D10_COMPARISON_NEVER
BorderColor[0]	1.0f
BorderColor[1]	1.0f
BorderColor[2]	1.0f
BorderColor[3]	1.0f
MinLOD	-FLT_MAX
MaxLOD	FLT_MAX

The method will not hold a reference to the interfaces passed in. For that reason, applications should be careful not to release an interface currently in use by the device.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::PSSetShader method (d3d10.h)

Article • 07/27/20222 minutes to read

Sets a pixel shader to the device.

Syntax

```
C++

void PSSetShader(
  [in] ID3D10PixelShader *pPixelShader
);
```

Parameters

`[in] pPixelShader`

Type: **ID3D10PixelShader***

Pointer to a pixel shader (see ID3D10PixelShader). Passing in **NULL** disables the shader for this pipeline stage.

Return value

None

Remarks

The method will not hold a reference to the interfaces passed in. For that reason, applications should be careful not to release an interface currently in use by the device.

Requirements

Target Platform	Windows

Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::PSSetShaderResources method (d3d10.h)

Article • 07/27/20222 minutes to read

Bind an array of shader resources to the pixel shader stage.

Syntax

C++

```
void PSSetShaderResources(
  [in] UINT                     StartSlot,
  [in] UINT                     NumViews,
  [in] ID3D10ShaderResourceView * const *ppShaderResourceViews
);
```

Parameters

`[in] StartSlot`

Type: **UINT**

Index into the device's zero-based array to begin setting shader resources to.

`[in] NumViews`

Type: **UINT**

Number of shader resources to set. Up to a maximum of 128 slots are available for shader resources.

`[in] ppShaderResourceViews`

Type: **ID3D10ShaderResourceView***

Array of shader resource view interfaces to set to the device.

Return value

None

Remarks

If you bind a subresource as an input and an output, this API will fill the destination shader resource slot with **NULL**. The debug layer (when active) will alert you if this is true.

For information about creating shader-resource views, see ID3D10Device::CreateShaderResourceView.

The method will not hold a reference to the interfaces passed in. For that reason, applications should be careful not to release an interface currently in use by the device.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::ResolveSubresource method (d3d10.h)

Article • 10/13/20212 minutes to read

Copy a multisampled resource into a non-multisampled resource. This API is most useful when re-using the resulting rendertarget of one render pass as an input to a second render pass.

Syntax

C++

```
void ResolveSubresource(
  [in] ID3D10Resource *pDstResource,
  [in] UINT           DstSubresource,
  [in] ID3D10Resource *pSrcResource,
  [in] UINT           SrcSubresource,
  [in] DXGI_FORMAT    Format
);
```

Parameters

`[in] pDstResource`

Type: **ID3D10Resource***

Destination resource. Must be a created with the D3D10_USAGE_DEFAULT flag and be single-sampled. See ID3D10Resource.

`[in] DstSubresource`

Type: **UINT**

A zero-based index, that identifies the destination subresource. See D3D10CalcSubresource for more details.

`[in] pSrcResource`

Type: **ID3D10Resource***

Source resource. Must be multisampled.

`[in] SrcSubresource`

Type: **UINT**

The source subresource of the source resource.

`[in] Format`

Type: **DXGI_FORMAT**

DXGI_FORMAT that indicates how the multisampled resource will be resolved to a single-sampled resource. See remarks.

Return value

None

Remarks

Both the source and destination resources must be the same resource type and have the same dimensions.

The source and destination must have compatible formats. There are three scenarios for this:

Scenario	Requirements
Source and destination are prestructured and typed	Both the source and destination must have identical formats and that format must be specified in the Format parameter.
One resource is prestructured and typed and the other is prestructured and typeless	The typed resource must have a format that is compatible with the typeless resource (i.e. the typed resource is DXGI_FORMAT_R32_FLOAT and the typeless resource is DXGI_FORMAT_R32_TYPELESS). The format of the typed resource must be specified in the Format parameter.
Source and destination are prestructured and typeless	Both the source and destination must have the same typeless format (i.e. both must have DXGI_FORMAT_R32_TYPELESS), and the Format parameter must specify a format that is compatible with the source and destination (i.e. if both are DXGI_FORMAT_R32_TYPELESS then DXGI_FORMAT_R32_FLOAT or DXGI_FORMAT_R32_UINT could be specified in the Format parameter).

Requirements

Target Platform	Windows
Header	d3d10.h (include D3d10core)

See also

ID3D10Device Interface

ID3D10Device::RSGetScissorRects method (d3d10.h)

Article • 07/27/20222 minutes to read

Get the array of scissor rectangles bound to the rasterizer stage.

Syntax

```
C++

void RSGetScissorRects(
  [in, out] UINT       *NumRects,
  [out]     D3D10_RECT *pRects
);
```

Parameters

`[in, out] NumRects`

Type: **UINT***

Number of scissor rectangles to get. If pRects is **NULL**, this will be filled with the number of scissor rectangles currently bound.

`[out] pRects`

Type: **D3D10_RECT***

An array of scissor rectangles (see D3D10_RECT). If NumRects is greater than the number of scissor rects currently bound, then unused members of the array will contain 0.

Return value

None

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::RSGetState method (d3d10.h)

Article • 07/27/20222 minutes to read

Get the rasterizer state from the rasterizer stage of the pipeline.

Syntax

```
C++

void RSGetState(
  [out] ID3D10RasterizerState **ppRasterizerState
);
```

Parameters

`[out] ppRasterizerState`

Type: **ID3D10RasterizerState****

Address of a pointer to a rasterizer-state interface (see ID3D10RasterizerState) to fill with information from the device.

Return value

None

Remarks

Any returned interfaces will have their reference count incremented by one. Applications should call IUnknown::Release on the returned interfaces when they are no longer needed to avoid memory leaks.

Requirements

Target Platform	Windows

Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::RSGetViewports method (d3d10.h)

Article • 10/13/20212 minutes to read

Get the array of viewports bound to the rasterizer stage

Syntax

C++

```
void RSGetViewports(
  [in, out] UINT           *NumViewports,
  [out]     D3D10_VIEWPORT *pViewports
);
```

Parameters

`[in, out] NumViewports`

Type: **UINT***

Number of viewports in *pViewports*.
If *pViewports* is **NULL**, this will be filled with the number of viewports currently bound.

`[out] pViewports`

Type: **D3D10_VIEWPORT***

An array of viewports (see D3D10_VIEWPORT) to be filled with information from the device. If NumViewports is greater than the actual number of viewports currently bound, then unused members of the array will contain 0.

Return value

None

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::RSSetScissorRects method (d3d10.h)

Article • 07/27/20222 minutes to read

Bind an array of scissor rectangles to the rasterizer stage.

Syntax

```
C++

void RSSetScissorRects(
  [in] UINT             NumRects,
  [in] const D3D10_RECT *pRects
);
```

Parameters

`[in] NumRects`

Type: **UINT**

Number of scissor rectangles to bind.

`[in] pRects`

Type: **const D3D10_RECT***

An array of scissor rectangles (see D3D10_RECT).

Return value

None

Remarks

The scissor rectangles will only be used if ScissorEnable is set to true in the rasterizer state (see D3D10_RASTERIZER_DESC).

Which scissor rectangle to use is determined by the SV_ViewportArrayIndex semantic output by a geometry shader (see shader semantic syntax). If a geometry shader does

not make use of the SV_ViewportArrayIndex semantic then Direct3D will use the first scissor rectangle in the array.

Each scissor rectangle in the array corresponds to a viewport in an array of viewports (see ID3D10Device::RSSetViewports).

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::RSSetState method (d3d10.h)

Article • 07/27/20222 minutes to read

Set the rasterizer state for the rasterizer stage of the pipeline.

Syntax

```
C++

void RSSetState(
  [in] ID3D10RasterizerState *pRasterizerState
);
```

Parameters

`[in] pRasterizerState`

Type: **ID3D10RasterizerState***

Pointer to a rasterizer-state interface (see ID3D10RasterizerState) to bind to the pipeline.

Return value

None

Remarks

To create a rasterizer state interface, call ID3D10Device::CreateRasterizerState. For more details on setting up the rasterizer state, see Set Rasterizer State.

The method will not hold a reference to the interfaces passed in. For that reason, applications should be careful not to release an interface currently in use by the device.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::RSSetViewports method (d3d10.h)

Article • 07/27/20222 minutes to read

Bind an array of viewports to the rasterizer stage of the pipeline.

Syntax

```
C++

void RSSetViewports(
  [in] UINT                 NumViewports,
  [in] const D3D10_VIEWPORT *pViewports
);
```

Parameters

`[in] NumViewports`

Type: **UINT**

Number of viewports to bind.

`[in] pViewports`

Type: **const D3D10_VIEWPORT***

An array of viewports (see D3D10_VIEWPORT) to bind to the device. Each viewport must have its extents within the allowed ranges: D3D10_VIEWPORT_BOUNDS_MIN, D3D10_VIEWPORT_BOUNDS_MAX, D3D10_MIN_DEPTH, and D3D10_MAX_DEPTH.

Return value

None

Remarks

All viewports must be set atomically as one operation. Any viewports not defined by the call are disabled.

Which viewport to use is determined by the SV_ViewportArrayIndex semantic output by a geometry shader (see shader semantic syntax). If a geometry shader does not make use of the SV_ViewportArrayIndex semantic then Direct3D will use the first viewport in the array.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::SetTextFilterSize method (d3d10.h)

Article • 07/27/20222 minutes to read

This method is not implemented.

Syntax

```
C++

void SetTextFilterSize(
  [in] UINT Width,
  [in] UINT Height
);
```

Parameters

`[in] Width`

Type: **UINT**

Not applicable

`[in] Height`

Type: **UINT**

Not applicable

Return value

None

Remarks

This method is not implemented, and should not be used.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::SOGetTargets method (d3d10.h)

Article • 10/13/20212 minutes to read

Get the target output buffers for the StreamOutput stage of the pipeline.

Syntax

C++

```
void SOGetTargets(
  [in]  UINT         NumBuffers,
  [out] ID3D10Buffer **ppSOTargets,
  [out] UINT         *pOffsets
);
```

Parameters

`[in] NumBuffers`

Type: **UINT**

Number of buffers to get. A maximum of four output buffers can be retrieved.

`[out] ppSOTargets`

Type: **ID3D10Buffer****

An array of output buffers (see ID3D10Buffer) to be retrieved from the device.

`[out] pOffsets`

Type: **UINT***

Array of offsets to the output buffers from *ppSOTargets*, one offset for each buffer. The offset values are in bytes.

Return value

None

Remarks

Any returned interfaces will have their reference count incremented by one. Applications should call IUnknown::Release on the returned interfaces when they are no longer needed to avoid memory leaks.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::SOSetTargets method (d3d10.h)

Article • 10/13/20212 minutes to read

Set the target output buffers for the StreamOutput stage, which enables/disables the pipeline to stream-out data.

Syntax

C++

```
void SOSetTargets(
  [in] UINT         NumBuffers,
  [in] ID3D10Buffer * const *ppSOTargets,
  [in] const UINT   *pOffsets
);
```

Parameters

`[in] NumBuffers`

Type: **UINT**

The number of buffer to bind to the device. A maximum of four output buffers can be set. If less than four are defined by the call, the remaining buffer slots are set to **NULL**. See Remarks.

`[in] ppSOTargets`

Type: **ID3D10Buffer***

The array of output buffers (see ID3D10Buffer) to bind to the device. The buffers must have been created with the D3D10_BIND_STREAM_OUTPUT flag.

`[in] pOffsets`

Type: **const UINT***

Array of offsets to the output buffers from *ppSOTargets*, one offset for each buffer. The offset values must be in bytes.

Return value

None

Remarks

Call **ID3D10Device::SOSetTargets** (before any draw calls) to stream data out; call SOSetTargets with **NULL** to stop streaming data out. For an example, see Exercise 01 from the GDC 2007 workshop, which sets the stream output rendertargets before calling draw methods in the RenderInstanceToStream function.

An offset of -1 will cause the stream output buffer to be appended, continuing after the last location written to the buffer in a previous stream output pass.

Calling this method using a buffer that is currently bound for writing will effectively bind **NULL** instead because a buffer cannot be bound as both an input and an output at the same time.

The Debug Layer will generate a warning whenever a resource is prevented from being bound simultaneously as an input and an output, but this will not prevent invalid data from being used by the runtime.

The method will not hold a reference to the interfaces passed in. For that reason, applications should be careful not to release an interface currently in use by the device.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::SetExceptionMode method (d3d10.h)

Article • 07/27/20222 minutes to read

Get the exception-mode flags.

Syntax

```
C++

HRESULT SetExceptionMode(
  [in] UINT RaiseFlags
);
```

Parameters

`[in] RaiseFlags`

Type: **UINT**

A value that contains one or more exception flags; each flag specifies a condition which will cause an exception to be raised. The flags are listed in D3D10_RAISE_FLAG. A default value of 0 means there are no flags.

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

Set an exception-mode flag to elevate an error condition to a non-continuable exception.

Whenever an error occurs, a Direct3D device enters the DEVICEREMOVED state and if the appropriate exception flag has been set, an exception is raised. A raised exception is designed to terminate an application. Before termination, the last chance an application has to persist data is by using an UnhandledExceptionFilter (see Structured Exception

Handling). In general, UnhandledExceptionFilters are leveraged to try to persist data when an application is crashing (to disk, for example). Any code that executes during an UnhandledExceptionFilter is not guaranteed to reliably execute (due to possible process corruption). Any data that the UnhandledExceptionFilter manages to persist, before the UnhandledExceptionFilter crashes again, should be treated as suspect, and therefore inspected by a new, non-corrupted process to see if it is usable.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::SetPredication method (d3d10.h)

Article • 07/27/20222 minutes to read

Set a rendering predicate.

Syntax

```
C++

void SetPredication(
  [in] ID3D10Predicate *pPredicate,
  [in] BOOL            PredicateValue
);
```

Parameters

`[in] pPredicate`

Type: **ID3D10Predicate***

Pointer to a predicate (see ID3D10Predicate). A **NULL** value indicates "no" predication; in this case, the value of PredicateValue is irrelevant but will be preserved for ID3D10Device::GetPredication.

`[in] PredicateValue`

Type: **BOOL**

If **TRUE**, rendering will be affected by when the predicate's conditions are met. If **FALSE**, rendering will be affected when the conditions are not met.

Return value

None

Remarks

The predicate must be in the "issued" or "signaled" state to be used for predication. While the predicate is set for predication, calls to ID3D10Asynchronous::Begin and

ID3D10Asynchronous::End are invalid.

This method is used to denote that subsequent rendering and resource manipulation commands are not actually performed if the resulting Predicate data of the Predicate is equal to the PredicateValue. However, some Predicates are only hints, so they may not actually prevent operations from being performed.

The primary usefulness of Predication is to allow an application to issue graphics commands without taking the performance hit of spinning, waiting for ID3D10Asynchronous::GetData to return. So, Predication can occur while **ID3D10Asynchronous::GetData** returns S_FALSE. Another way to think of it: an application can also use Predication as a fallback, if it is possible that **ID3D10Asynchronous::GetData** returns S_FALSE. If **ID3D10Asynchronous::GetData** returns S_OK, the application can skip calling the graphics commands manually with its own application logic.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::SetPrivateData method (d3d10.h)

Article07/27/2022

Set data to a device and associate that data with a guid.

Syntax

C++

```
HRESULT SetPrivateData(
  [in] REFGUID    guid,
  [in] UINT       DataSize,
  [in] const void *pData
);
```

Parameters

`[in] guid`

Type: **REFGUID**

Guid associated with the data.

`[in] DataSize`

Type: **UINT**

Size of the data.

`[in] pData`

Type: **const void***

Pointer to the data to be stored with this device. If pData is **NULL**, DataSize must also be 0, and any data previously associated with the guid will be destroyed.

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

The data stored in the device with this method can be retrieved with ID3D10DeviceChild::GetPrivateData.

The data and guid set with this method will typically be application-defined.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::SetPrivateDataInterface method (d3d10.h)

Article • 10/13/20212 minutes to read

Associate an IUnknown-derived interface with this device and associate that interface with an application-defined guid.

Syntax

```
C++

HRESULT SetPrivateDataInterface(
  [in] REFGUID        guid,
  [in] const IUnknown *pData
);
```

Parameters

`[in] guid`

Type: **REFGUID**

Guid associated with the interface.

`[in] pData`

Type: **const IUnknown***

Pointer to an IUnknown-derived interface to be associated with the device.

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

When this method is called ::addref() will be called on the IUnknown-derived interface, and when the device is destroyed ::release() will be called on the IUnknown-derived

interface.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::UpdateSubresource method (d3d10.h)

Article • 10/13/20214 minutes to read

The CPU copies data from memory to a subresource created in non-mappable memory. See remarks.

Syntax

```C++
void UpdateSubresource(
  [in] ID3D10Resource  *pDstResource,
  [in] UINT            DstSubresource,
  [in] const D3D10_BOX *pDstBox,
  [in] const void      *pSrcData,
  [in] UINT            SrcRowPitch,
  [in] UINT            SrcDepthPitch
);
```

Parameters

`[in] pDstResource`

Type: **ID3D10Resource***

A pointer to the destination resource (see ID3D10Resource Interface).

`[in] DstSubresource`

Type: **UINT**

A zero-based index, that identifies the destination subresource. See D3D10CalcSubresource for more details.

`[in] pDstBox`

Type: **const D3D10_BOX***

A box that defines the portion of the destination subresource to copy the resource data into. Coordinates are in bytes for buffers and in texels for textures. If **NULL**, the data is

written to the destination subresource with no offset. The dimensions of the source must fit the destination (see D3D10_BOX).

An empty box results in a no-op. A box is empty if the top value is greater than or equal to the bottom value, or the left value is greater than or equal to the right value, or the front value is greater than or equal to the back value. When the box is empty, **UpdateSubresource** doesn't perform an update operation.

`[in] pSrcData`

Type: **const void***

A pointer to the source data in memory.

`[in] SrcRowPitch`

Type: **UINT**

The size of one row of the source data.

`[in] SrcDepthPitch`

Type: **UINT**

The size of one depth slice of source data.

Return value

None

Remarks

For a shader-constant buffer; set pDstBox to **NULL**. It is not possible to use this method to partially update a shader-constant buffer.

A resource cannot be used as a destination if:

- the resource is created with immutable or dynamic usage.
- the resource is created as a depth-stencil resource.
- the resource is created with multisampling capability (see DXGI_SAMPLE_DESC).

When UpdateSubresource returns, the application is free to change or even free the data pointed to by pSrcData because the method has already copied/snapped away the original contents.

The performance of UpdateSubresource depends on whether or not there is contention for the destination resource. For example, contention for a vertex buffer resource occurs when the application executes a Draw call and later calls UpdateSubresource on the same vertex buffer before the Draw call is actually executed by the GPU.

- When there is contention for the resource, UpdateSubresource will perform 2 copies of the source data. First, the data is copied by the CPU to a temporary storage space accessible by the command buffer. This copy happens before the method returns. A second copy is then performed by the GPU to copy the source data into non-mappable memory. This second copy happens asynchronously because it is executed by GPU when the command buffer is flushed.
- When there is no resource contention, the behavior of UpdateSubresource is dependent on which is faster (from the CPU's perspective): copying the data to the command buffer and then having a second copy execute when the command buffer is flushed, or having the CPU copy the data to the final resource location. This is dependent on the architecture of the underlying system.

To better understand the source row pitch and source depth pitch parameters, consider the following illustration of a 3D volume texture.

Each block in this visual represents an element of data, and the size of each element is dependent on the resource's format. For example, if the resource format is DXGI_FORMAT_R32G32B32A32_FLOAT, then the size of each element would be 128 bits, or 16 bytes. This 3D volume texture has a width of two, a height of three, and a depth of four.

To calculate the source row pitch and source depth pitch for a given resource, use the following formulas:

- Source Row Pitch = [size of one element in bytes] * [number of elements in one row]
- Source Depth Pitch = [Source Row Pitch] * [number of rows (height)]

In the case of this example 3D volume texture where the size of each element is 16 bytes, the formulas are as follows:

- Source Row Pitch = 16 * 2 = 32
- Source Depth Pitch = 16 * 2 * 3 = 96

The following illustration shows the resource as it is laid out in memory.

For example, the following code snippet shows how to specify a destination region in a 2D texture. Assume the destination texture is 512x512 and the operation will copy the data pointed to by pData to [(120,100)..(200,220)] in the destination texture. Also assume that rowPitch has been initialized with the proper value (as explained above). Front and back are set to 0 and 1 respectively, because by having front equal to back, the box is technically empty.

```
D3D10_BOX destRegion;
destRegion.left = 120;
destRegion.right = 200;
destRegion.top = 100;
destRegion.bottom = 220;
destRegion.front = 0;
destRegion.back = 1;

pd3dDevice->UpdateSubresource( pDestTexture, 0, &destRegion, pData,
rowPitch, 0 );
```

The 1D case is similar. The following snippet shows how to specify a destination region in a 1D texture. Use the same assumptions as above, except that the texture is 512 in length.

```
D3D10_BOX destRegion;
destRegion.left = 120;
destRegion.right = 200;
destRegion.top = 0;
destRegion.bottom = 1;
destRegion.front = 0;
destRegion.back = 1;

pd3dDevice->UpdateSubresource( pDestTexture, 0, &destRegion, pData,
rowPitch, 0 );
```

Differences between Direct3D 10 and Direct3D 10.1:
Direct3D 10.1 enables depth-stencil resources to be used as either a source or destination.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device

ID3D10Resource Interface

ID3D10Device::VSGetConstantBuffers method (d3d10.h)

Article • 07/27/20222 minutes to read

Get the constant buffers used by the vertex shader pipeline stage.

Syntax

```
C++

void VSGetConstantBuffers(
  [in]  UINT         StartSlot,
  [in]  UINT         NumBuffers,
  [out] ID3D10Buffer **ppConstantBuffers
);
```

Parameters

`[in] StartSlot`

Type: **UINT**

Index into the device's zero-based array to begin retrieving constant buffers from.

`[in] NumBuffers`

Type: **UINT**

Number of buffers to retrieve.

`[out] ppConstantBuffers`

Type: **ID3D10Buffer****

Array of constant buffer interface pointers (see ID3D10Buffer) to be returned by the method.

Return value

None

Remarks

Any returned interfaces will have their reference count incremented by one. Applications should call IUnknown::Release on the returned interfaces when they are no longer needed to avoid memory leaks.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::VSGetSamplers method (d3d10.h)

Article • 07/27/20222 minutes to read

Get an array of sampler states from the vertex shader pipeline stage.

Syntax

```C++
void VSGetSamplers(
  [in]  UINT               StartSlot,
  [in]  UINT               NumSamplers,
  [out] ID3D10SamplerState **ppSamplers
);
```

Parameters

`[in] StartSlot`

Type: **UINT**

Index into the device's zero-based array to begin getting samplers from.

`[in] NumSamplers`

Type: **UINT**

Number of samplers to get from the device. Each pipeline stage has a total of 16 sampler slots available.

`[out] ppSamplers`

Type: **ID3D10SamplerState****

Array of sampler-state interface pointers (see ID3D10SamplerState) to be returned by the device.

Return value

None

Remarks

Any returned interfaces will have their reference count incremented by one. Applications should call IUnknown::Release on the returned interfaces when they are no longer needed to avoid memory leaks.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::VSGetShader method (d3d10.h)

Article • 07/27/20222 minutes to read

Get the vertex shader currently set on the device.

Syntax

```
C++

void VSGetShader(
  [out] ID3D10VertexShader **ppVertexShader
);
```

Parameters

`[out] ppVertexShader`

Type: **ID3D10VertexShader****

Address of a pointer to a vertex shader (see ID3D10VertexShader) to be returned by the method.

Return value

None

Remarks

Any returned interfaces will have their reference count incremented by one. Applications should call IUnknown::Release on the returned interfaces when they are no longer needed to avoid memory leaks.

Requirements

Target Platform	Windows

Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::VSGetShaderResources method (d3d10.h)

Article • 07/27/20222 minutes to read

Get the vertex shader resources.

Syntax

```
C++

void VSGetShaderResources(
  [in]  UINT                     StartSlot,
  [in]  UINT                     NumViews,
  [out] ID3D10ShaderResourceView **ppShaderResourceViews
);
```

Parameters

`[in] StartSlot`

Type: **UINT**

Index into the device's zero-based array to begin getting shader resources from.

`[in] NumViews`

Type: **UINT**

The number of resources to get from the device. Up to a maximum of 128 slots are available for shader resources.

`[out] ppShaderResourceViews`

Type: **ID3D10ShaderResourceView****

Array of shader resource view interfaces to be returned by the device.

Return value

None

Remarks

Any returned interfaces will have their reference count incremented by one. Applications should call IUnknown::Release on the returned interfaces when they are no longer needed to avoid memory leaks.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::VSSetConstantBuffers method (d3d10.h)

Article10/13/2021

Set the constant buffers used by the vertex shader pipeline stage.

Syntax

C++

```
void VSSetConstantBuffers(
  [in] UINT         StartSlot,
  [in] UINT         NumBuffers,
  [in] ID3D10Buffer * const *ppConstantBuffers
);
```

Parameters

`[in] StartSlot`

Type: **UINT**

Index into the device's zero-based array to begin setting constant buffers to.

`[in] NumBuffers`

Type: **UINT**

Number of buffers to set.

`[in] ppConstantBuffers`

Type: **ID3D10Buffer***

Array of constant buffers (see ID3D10Buffer) being given to the device.

Return value

None

Remarks

The method will not hold a reference to the interfaces passed in. For that reason, applications should be careful not to release an interface currently in use by the device.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::VSSetSamplers method (d3d10.h)

Article07/27/2022

Set an array of sampler states to the vertex shader pipeline stage.

Syntax

```
C++

void VSSetSamplers(
  [in] UINT               StartSlot,
  [in] UINT               NumSamplers,
  [in] ID3D10SamplerState * const *ppSamplers
);
```

Parameters

`[in] StartSlot`

Type: **UINT**

Index into the device's zero-based array to begin setting samplers to.

`[in] NumSamplers`

Type: **UINT**

Number of samplers in the array. Each pipeline stage has a total of 16 sampler slots available.

`[in] ppSamplers`

Type: **ID3D10SamplerState***

Pointer to an array of sampler-state interfaces (see ID3D10SamplerState). See Remarks.

Return value

None

Remarks

Any sampler may be set to **NULL**; this invokes the default state, which is defined to be the following.

```
//Default sampler state:
D3D10_SAMPLER_DESC SamplerDesc;
SamplerDesc.Filter = D3D10_FILTER_MIN_MAG_MIP_LINEAR;
SamplerDesc.AddressU = D3D10_TEXTURE_ADDRESS_CLAMP;
SamplerDesc.AddressV = D3D10_TEXTURE_ADDRESS_CLAMP;
SamplerDesc.AddressW = D3D10_TEXTURE_ADDRESS_CLAMP;
SamplerDesc.MipLODBias = 0;
SamplerDesc.MaxAnisotropy = 1;
SamplerDesc.ComparisonFunc = D3D10_COMPARISON_NEVER;
SamplerDesc.BorderColor[0] = 1.0f;
SamplerDesc.BorderColor[1] = 1.0f;
SamplerDesc.BorderColor[2] = 1.0f;
SamplerDesc.BorderColor[3] = 1.0f;
SamplerDesc.MinLOD = -FLT_MAX;
SamplerDesc.MaxLOD = FLT_MAX;
```

The method will not hold a reference to the interfaces passed in. For that reason, applications should be careful not to release an interface currently in use by the device.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::VSSetShader method (d3d10.h)

Article • 07/27/20222 minutes to read

Set a vertex shader to the device.

Syntax

```
C++

void VSSetShader(
  [in] ID3D10VertexShader *pVertexShader
);
```

Parameters

`[in] pVertexShader`

Type: **ID3D10VertexShader***

Pointer to a vertex shader (see ID3D10VertexShader). Passing in **NULL** disables the shader for this pipeline stage.

Return value

None

Remarks

The method will not hold a reference to the interfaces passed in. For that reason, applications should be careful not to release an interface currently in use by the device.

Requirements

Target Platform	Windows

Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device::VSSetShaderResources method (d3d10.h)

Article • 10/13/20212 minutes to read

Bind an array of shader resources to the vertex shader stage.

Syntax

C++

```
void VSSetShaderResources(
  [in] UINT                     StartSlot,
  [in] UINT                     NumViews,
  [in] ID3D10ShaderResourceView * const *ppShaderResourceViews
);
```

Parameters

`[in] StartSlot`

Type: **UINT**

Index into the device's zero-based array to begin setting shader resources to.

`[in] NumViews`

Type: **UINT**

Number of shader resources to set. Up to a maximum of 128 slots are available for shader resources.

`[in] ppShaderResourceViews`

Type: **ID3D10ShaderResourceView***

Array of shader resource view interfaces to set to the device.

Return value

None

Remarks

If you bind a subresource as an input and an output, this API will fill the destination shader resource slot with **NULL**. The debug layer (when active) will alert you if this is true.

For information about creating shader-resource views, see ID3D10Device::CreateShaderResourceView.

The method will not hold a reference to the interfaces passed in. For that reason, applications should be careful not to release an interface currently in use by the device.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Device1 interface (d3d10_1.h)

Article07/27/2022

The device interface represents a virtual adapter for Direct3D 10.1; it is used to perform rendering and create Direct3D resources.

Inheritance

The **ID3D10Device1** interface inherits from ID3D10Device. **ID3D10Device1** also has these types of members:

Methods

The **ID3D10Device1** interface has these methods.

ID3D10Device1::CreateBlendState1 Create a blend-state object that encapsulates blend state for the output-merger stage. (ID3D10Device1.CreateBlendState1)
ID3D10Device1::CreateShaderResourceView1 Create a shader-resource view for accessing data in a resource. (ID3D10Device1.CreateShaderResourceView1)
ID3D10Device1::GetFeatureLevel Gets the feature level of the hardware device. (ID3D10Device1.GetFeatureLevel)

Remarks

A device is created using D3D10CreateDevice1.

This method requires Windows Vista Service Pack 1.

Requirements

Target Platform	Windows
Header	d3d10_1.h

See also

Core Interfaces

ID3D10Device

ID3D10Device1::CreateBlendState1 method (d3d10_1.h)

Article • 07/27/20222 minutes to read

Create a blend-state object that encapsulates blend state for the output-merger stage.

Syntax

C++

```
HRESULT CreateBlendState1(
  [in]  const D3D10_BLEND_DESC1 *pBlendStateDesc,
  [out] ID3D10BlendState1       **ppBlendState
);
```

Parameters

`[in] pBlendStateDesc`

Type: **const D3D10_BLEND_DESC1***

Pointer to a blend-state description (see D3D10_BLEND_DESC1).

`[out] ppBlendState`

Type: **ID3D10BlendState1****

Address of a pointer to the blend-state object created (see ID3D10BlendState1 Interface).

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

An application can create up to 4096 unique blend-state objects. For each object created, the runtime checks to see if a previous object has the same state. If such a

previous object exists, the runtime will return a pointer to previous instance instead of creating a duplicate object.

This method requires Windows Vista Service Pack 1.

Requirements

Target Platform	Windows
Header	d3d10_1.h

See also

ID3D10Device1 Interface

ID3D10Device1::CreateShaderResourceView1 method (d3d10_1.h)

Article • 07/27/20222 minutes to read

Create a shader-resource view for accessing data in a resource.

Syntax

```
C++

HRESULT CreateShaderResourceView1(
  [in]  ID3D10Resource                        *pResource,
  [in]  const D3D10_SHADER_RESOURCE_VIEW_DESC1 *pDesc,
  [out] ID3D10ShaderResourceView1             **ppSRView
);
```

Parameters

`[in] pResource`

Type: **ID3D10Resource***

Pointer to the resource that will serve as input to a shader. This resource must have been created with the D3D10_BIND_SHADER_RESOURCE flag.

`[in] pDesc`

Type: **const D3D10_SHADER_RESOURCE_VIEW_DESC1***

Pointer to a shader-resource-view description (see D3D10_SHADER_RESOURCE_VIEW_DESC1). Set this parameter to **NULL** to create a view that accesses the entire resource (using the format the resource was created with).

`[out] ppSRView`

Type: **ID3D10ShaderResourceView1****

Address of a pointer to a shader-resource view (see ID3D10ShaderResourceView1 Interface). Set this parameter to **NULL** to validate the other input parameters (the method will return S_FALSE if the other input parameters pass validation).

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

A resource is made up of one or more subresources, a view identifies which subresources to allow the pipeline to access. In addition, each resource is bound to the pipeline using a view. A shader-resource view is designed to bind any buffer or texture resource to the shader stages using the following API methods: VSSetShaderResources, GSSetShaderResources and PSSetShaderResources.

Since a view is fully typed, this means that typeless resources become fully typed when bound to the pipeline.

This method requires Windows Vista Service Pack 1.

Requirements

Target Platform	Windows
Header	d3d10_1.h

See also

ID3D10Device1 Interface

ID3D10Device1::GetFeatureLevel method (d3d10_1.h)

Article07/27/2022

Gets the feature level of the hardware device.

Syntax

```
C++

D3D10_FEATURE_LEVEL1 GetFeatureLevel();
```

Return value

Type: **D3D10_FEATURE_LEVEL1**

The feature level (see D3D10_FEATURE_LEVEL1).

Remarks

This method requires Windows Vista Service Pack 1.

Requirements

Target Platform	Windows
Header	d3d10_1.h

See also

ID3D10Device1 Interface

ID3D10DeviceChild interface (d3d10.h)

Article07/27/2022

A device-child interface accesses data used by a device.

Inheritance

The **ID3D10DeviceChild** interface inherits from the IUnknown interface. **ID3D10DeviceChild** also has these types of members:

Methods

The **ID3D10DeviceChild** interface has these methods.

ID3D10DeviceChild::GetDevice Get a pointer to the device that created this interface. (ID3D10DeviceChild.GetDevice)
ID3D10DeviceChild::GetPrivateData Get application-defined data from a device child. (ID3D10DeviceChild.GetPrivateData)
ID3D10DeviceChild::SetPrivateData Set application-defined data to a device child and associate that data with an application-defined guid. (ID3D10DeviceChild.SetPrivateData)
ID3D10DeviceChild::SetPrivateDataInterface Associate an IUnknown-derived interface with this device child and associate that interface with an application-defined guid. (ID3D10DeviceChild.SetPrivateDataInterface)

Remarks

There are several types of device child interfaces, all of which inherit this interface. They include shaders, state objects, and input layouts.

Requirements

Target Platform	Windows
Header	d3d10.h

See also

Core Interfaces

ID3D10DeviceChild::GetDevice method (d3d10.h)

Article • 07/27/20222 minutes to read

Get a pointer to the device that created this interface.

Syntax

C++

```
void GetDevice(
  [out] ID3D10Device **ppDevice
);
```

Parameters

`[out] ppDevice`

Type: **ID3D10Device****

Address of a pointer to a device (see ID3D10Device Interface).

Return value

None

Remarks

Any returned interfaces will have their reference count incremented by one, so be sure to call ::release() on the returned pointer(s) before they are freed or else you will have a memory leak.

Requirements

Target Platform	Windows

Header	d3d10.h
Library	D3D10.lib

See also

ID3D10DeviceChild Interface

ID3D10DeviceChild::GetPrivateData method (d3d10.h)

Article • 07/27/20222 minutes to read

Get application-defined data from a device child.

Syntax

C++

```
HRESULT GetPrivateData(
  [in]      REFGUID guid,
  [in, out] UINT    *pDataSize,
  [out]     void    *pData
);
```

Parameters

`[in] guid`

Type: **REFGUID**

Guid associated with the data.

`[in, out] pDataSize`

Type: **UINT***

Size of the data.

`[out] pData`

Type: **void***

Pointer to the data stored with the device child. If pData is **NULL**, DataSize must also be 0, and any data previously associated with the guid will be destroyed.

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

The data stored in the device child is set with ID3D10DeviceChild::SetPrivateData.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10DeviceChild Interface

ID3D10DeviceChild::SetPrivateData method (d3d10.h)

Article • 07/27/20222 minutes to read

Set application-defined data to a device child and associate that data with an application-defined guid.

Syntax

```
C++

HRESULT SetPrivateData(
  [in] REFGUID    guid,
  [in] UINT       DataSize,
  [in] const void *pData
);
```

Parameters

`[in] guid`

Type: **REFGUID**

Guid associated with the data.

`[in] DataSize`

Type: **UINT**

Size of the data.

`[in] pData`

Type: **const void***

Pointer to the data to be stored with this device child. If pData is **NULL**, DataSize must also be 0, and any data previously associated with the specified guid will be destroyed.

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

The data stored in the device child with this method can be retrieved with ID3D10DeviceChild::GetPrivateData.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10DeviceChild Interface

ID3D10DeviceChild::SetPrivateDataInterface method (d3d10.h)

Article07/27/2022

Associate an IUnknown-derived interface with this device child and associate that interface with an application-defined guid.

Syntax

```
C++

HRESULT SetPrivateDataInterface(
  [in] REFGUID        guid,
  [in] const IUnknown *pData
);
```

Parameters

`[in] guid`

Type: **REFGUID**

Guid associated with the interface.

`[in] pData`

Type: **const IUnknown***

Pointer to an IUnknown-derived interface to be associated with the device child.

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

When this method is called ::addref() will be called on the IUnknown-derived interface, and when the device child is destroyed ::release() will be called on the IUnknown-

derived interface.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10DeviceChild Interface

ID3D10InfoQueue interface (d3d10sdklayers.h)

Article • 07/27/20223 minutes to read

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.

Inheritance

The **ID3D10InfoQueue** interface inherits from the IUnknown interface. **ID3D10InfoQueue** also has these types of members:

Methods

The **ID3D10InfoQueue** interface has these methods.

ID3D10InfoQueue::AddApplicationMessage Add a user-defined message to the message queue and send that message to debug output. (ID3D10InfoQueue.AddApplicationMessage)
ID3D10InfoQueue::AddMessage Add a Direct3D 10 debug message to the message queue and send that message to debug output.
ID3D10InfoQueue::AddRetrievalFilterEntries Add storage filters to the top of the retrieval-filter stack. (ID3D10InfoQueue.AddRetrievalFilterEntries)
ID3D10InfoQueue::AddStorageFilterEntries Add storage filters to the top of the storage-filter stack. (ID3D10InfoQueue.AddStorageFilterEntries)
ID3D10InfoQueue::ClearRetrievalFilter Remove a retrieval filter from the top of the retrieval-filter stack. (ID3D10InfoQueue.ClearRetrievalFilter)

ID3D10InfoQueue::ClearStorageFilter Remove a storage filter from the top of the storage-filter stack. (ID3D10InfoQueue.ClearStorageFilter)
ID3D10InfoQueue::ClearStoredMessages Clear all messages from the message queue. (ID3D10InfoQueue.ClearStoredMessages)
ID3D10InfoQueue::GetBreakOnCategory Get a message category to break on when a message with that category passes through the storage filter. (ID3D10InfoQueue.GetBreakOnCategory)
ID3D10InfoQueue::GetBreakOnID Get a message identifier to break on when a message with that identifier passes through the storage filter. (ID3D10InfoQueue.GetBreakOnID)
ID3D10InfoQueue::GetBreakOnSeverity Get a message severity level to break on when a message with that severity level passes through the storage filter. (ID3D10InfoQueue.GetBreakOnSeverity)
ID3D10InfoQueue::GetMessage Get a message from the message queue. (ID3D10InfoQueue.GetMessage)
ID3D10InfoQueue::GetMessageCountLimit Get the maximum number of messages that can be added to the message queue. (ID3D10InfoQueue.GetMessageCountLimit)
ID3D10InfoQueue::GetMuteDebugOutput Get a boolean that turns the debug output on or off. (ID3D10InfoQueue.GetMuteDebugOutput)
ID3D10InfoQueue::GetNumMessagesAllowedByStorageFilter Get the number of messages that were allowed to pass through a storage filter. (ID3D10InfoQueue.GetNumMessagesAllowedByStorageFilter)
ID3D10InfoQueue::GetNumMessagesDeniedByStorageFilter Get the number of messages that were denied passage through a storage filter. (ID3D10InfoQueue.GetNumMessagesDeniedByStorageFilter)

ID3D10InfoQueue::GetNumMessagesDiscardedByMessageCountLimit Get the number of messages that were discarded due to the message count limit. (ID3D10InfoQueue.GetNumMessagesDiscardedByMessageCountLimit)
ID3D10InfoQueue::GetNumStoredMessages Get the number of messages currently stored in the message queue. (ID3D10InfoQueue.GetNumStoredMessages)
ID3D10InfoQueue::GetNumStoredMessagesAllowedByRetrievalFilter Get the number of messages that are able to pass through a retrieval filter. (ID3D10InfoQueue.GetNumStoredMessagesAllowedByRetrievalFilter)
ID3D10InfoQueue::GetRetrievalFilter Get the retrieval filter at the top of the retrieval-filter stack. (ID3D10InfoQueue.GetRetrievalFilter)
ID3D10InfoQueue::GetRetrievalFilterStackSize Get the size of the retrieval-filter stack in bytes. (ID3D10InfoQueue.GetRetrievalFilterStackSize)
ID3D10InfoQueue::GetStorageFilter Get the storage filter at the top of the storage-filter stack. (ID3D10InfoQueue.GetStorageFilter)
ID3D10InfoQueue::GetStorageFilterStackSize Get the size of the storage-filter stack in bytes. (ID3D10InfoQueue.GetStorageFilterStackSize)
ID3D10InfoQueue::PopRetrievalFilter Pop a retrieval filter from the top of the retrieval-filter stack. (ID3D10InfoQueue.PopRetrievalFilter)
ID3D10InfoQueue::PopStorageFilter Pop a storage filter from the top of the storage-filter stack. (ID3D10InfoQueue.PopStorageFilter)
ID3D10InfoQueue::PushCopyOfRetrievalFilter Push a copy of retrieval filter currently on the top of the retrieval-filter stack onto the retrieval-filter stack. (ID3D10InfoQueue.PushCopyOfRetrievalFilter)
ID3D10InfoQueue::PushCopyOfStorageFilter Push a copy of storage filter currently on the top of the storage-filter stack onto the storage-filter stack. (ID3D10InfoQueue.PushCopyOfStorageFilter)

ID3D10InfoQueue::PushEmptyRetrievalFilter Push an empty retrieval filter onto the retrieval-filter stack. (ID3D10InfoQueue.PushEmptyRetrievalFilter)
ID3D10InfoQueue::PushEmptyStorageFilter Push an empty storage filter onto the storage-filter stack. (ID3D10InfoQueue.PushEmptyStorageFilter)
ID3D10InfoQueue::PushRetrievalFilter Push a retrieval filter onto the retrieval-filter stack. (ID3D10InfoQueue.PushRetrievalFilter)
ID3D10InfoQueue::PushStorageFilter Push a storage filter onto the storage-filter stack. (ID3D10InfoQueue.PushStorageFilter)
ID3D10InfoQueue::SetBreakOnCategory Set a message category to break on when a message with that category passes through the storage filter. (ID3D10InfoQueue.SetBreakOnCategory)
ID3D10InfoQueue::SetBreakOnID Set a message identifier to break on when a message with that identifier passes through the storage filter. (ID3D10InfoQueue.SetBreakOnID)
ID3D10InfoQueue::SetBreakOnSeverity Set a message severity level to break on when a message with that severity level passes through the storage filter. (ID3D10InfoQueue.SetBreakOnSeverity)
ID3D10InfoQueue::SetMessageCountLimit Set the maximum number of messages that can be added to the message queue. (ID3D10InfoQueue.SetMessageCountLimit)
ID3D10InfoQueue::SetMuteDebugOutput Set a boolean that turns the debug output on or off. (ID3D10InfoQueue.SetMuteDebugOutput)

Remarks

This interface is obtained by turning on the debug layer and querying it from the ID3D10Device Interface using IUnknown::QueryInterface.

```
hr = D3D10CreateDeviceAndSwapChain( NULL, g_driverType, NULL,
D3D10_CREATE_DEVICE_DEBUG, D3D10_SDK_VERSION, &sd, &g_pSwapChain,
&g_pd3dDevice );
...
ID3D10InfoQueue * infoQueue;
g_pd3dDevice->QueryInterface(__uuidof(ID3D10InfoQueue),  (void
**)&infoQueue);
```

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

Core Interfaces

ID3D10InfoQueue::AddApplicationMessage method (d3d10sdklayers.h)

Article • 07/27/20222 minutes to read

Add a user-defined message to the message queue and send that message to debug output.

Syntax

```
C++

HRESULT AddApplicationMessage(
  [in] D3D10_MESSAGE_SEVERITY Severity,
  [in] LPCSTR                 pDescription
);
```

Parameters

`[in] Severity`

Type: **D3D10_MESSAGE_SEVERITY**

Severity of a message (see D3D10_MESSAGE_SEVERITY).

`[in] pDescription`

Type: **LPCSTR**

Message string.

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10InfoQueue Interface

ID3D10InfoQueue::AddMessage method (d3d10sdklayers.h)

Article10/13/2021

Add a Direct3D 10 debug message to the message queue and send that message to debug output.

Syntax

```
C++

HRESULT AddMessage(
  [in] D3D10_MESSAGE_CATEGORY Category,
  [in] D3D10_MESSAGE_SEVERITY Severity,
  [in] D3D10_MESSAGE_ID       ID,
  [in] LPCSTR                 pDescription
);
```

Parameters

`[in] Category`

Type: **D3D10_MESSAGE_CATEGORY**

Category of a message (see D3D10_MESSAGE_CATEGORY).

`[in] Severity`

Type: **D3D10_MESSAGE_SEVERITY**

Severity of a message (see D3D10_MESSAGE_SEVERITY).

`[in] ID`

Type: **D3D10_MESSAGE_ID**

Unique identifier of a message (see D3D10_MESSAGE_ID).

`[in] pDescription`

Type: **LPCSTR**

User-defined message.

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

This method is used by the runtime's internal mechanisms to add Direct3D 10 debug messages to the message queue and send them to debug output. For applications to add their own custom messages to the message queue and send them to debug output, call ID3D10InfoQueue::AddApplicationMessage.

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10InfoQueue Interface

ID3D10InfoQueue::AddRetrievalFilterEntries method (d3d10sdklayers.h)

Article • 07/27/20222 minutes to read

Add storage filters to the top of the retrieval-filter stack.

Syntax

```
C++

HRESULT AddRetrievalFilterEntries(
  [in] D3D10_INFO_QUEUE_FILTER *pFilter
);
```

Parameters

`[in] pFilter`

Type: **D3D10_INFO_QUEUE_FILTER***

Array of retrieval filters (see D3D10_INFO_QUEUE_FILTER).

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

A retrieval filter is used to define a subgroup of the messages that are already in the info queue.
Retrieval filters affect the messages that will be returned by ID3D10InfoQueue::GetMessage.

The number of messages already in the info queue that will be allowed through the retrieval filter can be determined by calling ID3D10InfoQueue::GetNumStoredMessagesAllowedByRetrievalFilter.

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10InfoQueue Interface

ID3D10InfoQueue::AddStorageFilterEntries method (d3d10sdklayers.h)

Article • 07/27/20222 minutes to read

Add storage filters to the top of the storage-filter stack.

Syntax

```
C++

HRESULT AddStorageFilterEntries(
  [in] D3D10_INFO_QUEUE_FILTER *pFilter
);
```

Parameters

`[in] pFilter`

Type: **D3D10_INFO_QUEUE_FILTER***

Array of storage filters (see D3D10_INFO_QUEUE_FILTER).

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

A storage filter defines a grouping of debug messages that should be allowed into the info queue.

Requirements

Target Platform	Windows

Header	d3d10sdklayers.h

See also

ID3D10InfoQueue Interface

ID3D10InfoQueue::ClearRetrievalFilter method (d3d10sdklayers.h)

Article • 07/27/20222 minutes to read

Remove a retrieval filter from the top of the retrieval-filter stack.

Syntax

C++

```
void ClearRetrievalFilter();
```

Return value

None

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10InfoQueue Interface

ID3D10InfoQueue::ClearStorageFilter method (d3d10sdklayers.h)

Article • 07/27/20222 minutes to read

Remove a storage filter from the top of the storage-filter stack.

Syntax

```
C++

void ClearStorageFilter();
```

Return value

None

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10InfoQueue Interface

ID3D10InfoQueue::ClearStoredMessages method (d3d10sdklayers.h)

Article • 07/27/20222 minutes to read

Clear all messages from the message queue.

Syntax

```
C++

void ClearStoredMessages();
```

Return value

None

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10InfoQueue Interface

ID3D10InfoQueue::GetBreakOnCategory method (d3d10sdklayers.h)

Article • 07/27/20222 minutes to read

Get a message category to break on when a message with that category passes through the storage filter.

Syntax

```
C++

BOOL GetBreakOnCategory(
  [in] D3D10_MESSAGE_CATEGORY Category
);
```

Parameters

`[in] Category`

Type: **D3D10_MESSAGE_CATEGORY**

Message category to break on (see D3D10_MESSAGE_CATEGORY).

Return value

Type: **BOOL**

Whether this breaking condition is turned on or off (true for on, false for off).

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10InfoQueue Interface

ID3D10InfoQueue::GetBreakOnID method (d3d10sdklayers.h)

Article • 07/27/20222 minutes to read

Get a message identifier to break on when a message with that identifier passes through the storage filter.

Syntax

```
C++

BOOL GetBreakOnID(
  [in] D3D10_MESSAGE_ID ID
);
```

Parameters

`[in] ID`

Type: **D3D10_MESSAGE_ID**

Message identifier to break on (see D3D10_MESSAGE_ID).

Return value

Type: **BOOL**

Whether this breaking condition is turned on or off (true for on, false for off).

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10InfoQueue Interface

ID3D10InfoQueue::GetBreakOnSeverity method (d3d10sdklayers.h)

Article • 07/27/20222 minutes to read

Get a message severity level to break on when a message with that severity level passes through the storage filter.

Syntax

```
C++

BOOL GetBreakOnSeverity(
  [in] D3D10_MESSAGE_SEVERITY Severity
);
```

Parameters

`[in] Severity`

Type: **D3D10_MESSAGE_SEVERITY**

Message severity level to break on (see D3D10_MESSAGE_SEVERITY).

Return value

Type: **BOOL**

Whether this breaking condition is turned on or off (true for on, false for off).

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10InfoQueue Interface

ID3D10InfoQueue::GetMessage method (d3d10sdklayers.h)

Article • 07/27/20222 minutes to read

Get a message from the message queue.

Syntax

```
C++

HRESULT GetMessage(
  [in]      UINT64        MessageIndex,
  [out]     D3D10_MESSAGE *pMessage,
  [in, out] SIZE_T        *pMessageByteLength
);
```

Parameters

`[in] MessageIndex`

Type: **UINT64**

Index into message queue after an optional retrieval filter has been applied. This can be between 0 and the number of messages in the message queue that pass through the retrieval filter (which can be obtained with ID3D10InfoQueue::GetNumStoredMessagesAllowedByRetrievalFilter). 0 is the message at the front of the message queue.

`[out] pMessage`

Type: **D3D10_MESSAGE***

Returned message (see D3D10_MESSAGE).

`[in, out] pMessageByteLength`

Type: **SIZE_T***

Size of pMessage in bytes, including the size of the message string that the pMessage points to.

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

This method does not remove any messages from the message queue.

This method gets messages from the message queue after an optional retrieval filter has been applied.

Applications should call this method twice to retrieve a message - first to obtain the size of the message and second to get the message. Here is a typical example:

```
// Get the size of the message
SIZE_T messageLength = 0;
HRESULT hr = pInfoQueue->GetMessage(0, NULL, &messageLength);

// Allocate space and get the message
D3D10_MESSAGE * pMessage = (D3D10_MESSAGE*)malloc(messageLength);
hr = pInfoQueue->GetMessage(0, pMessage, &messageLength);
```

For an overview see Information Queue Overview.

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10InfoQueue Interface

ID3D10InfoQueue::GetMessageCountLimit method (d3d10sdklayers.h)

Article07/27/2022

Get the maximum number of messages that can be added to the message queue.

Syntax

```
C++

UINT64 GetMessageCountLimit();
```

Return value

Type: **UINT64**

Maximum number of messages that can be added to the queue. -1 means no limit.

Remarks

When the number of messages in the message queue has reached the maximum limit, new messages coming in will push old messages out.

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10InfoQueue Interface

ID3D10InfoQueue::GetMuteDebugOutput method (d3d10sdklayers.h)

Article • 07/27/20222 minutes to read

Get a boolean that turns the debug output on or off.

Syntax

```
C++

BOOL GetMuteDebugOutput();
```

Return value

Type: **BOOL**

Whether the debug output is on or off (true for on, false for off).

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10InfoQueue Interface

ID3D10InfoQueue::GetNumMessagesAllowedByStorageFilter method (d3d10sdklayers.h)

Article • 07/27/20222 minutes to read

Get the number of messages that were allowed to pass through a storage filter.

Syntax

```
C++

UINT64 GetNumMessagesAllowedByStorageFilter();
```

Return value

Type: **UINT64**

Number of messages allowed by a storage filter.

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10InfoQueue Interface

ID3D10InfoQueue::GetNumMessagesDeniedByStorageFilter method (d3d10sdklayers.h)

Article • 07/27/20222 minutes to read

Get the number of messages that were denied passage through a storage filter.

Syntax

```
C++

UINT64 GetNumMessagesDeniedByStorageFilter();
```

Return value

Type: **UINT64**

Number of messages denied by a storage filter.

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10InfoQueue Interface

ID3D10InfoQueue::GetNumMessagesDiscardedByMessageCountLimit method (d3d10sdklayers.h)

Article • 07/27/20222 minutes to read

Get the number of messages that were discarded due to the message count limit.

Syntax

```
C++

UINT64 GetNumMessagesDiscardedByMessageCountLimit();
```

Return value

Type: **UINT64**

Number of messages discarded.

Remarks

Get and set the message count limit with ID3D10InfoQueue::GetMessageCountLimit and ID3D10InfoQueue::SetMessageCountLimit, respectively.

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10InfoQueue Interface

ID3D10InfoQueue::GetNumStoredMessages method (d3d10sdklayers.h)

Article • 07/27/20222 minutes to read

Get the number of messages currently stored in the message queue.

Syntax

```
C++

UINT64 GetNumStoredMessages();
```

Return value

Type: **UINT64**

Number of messages currently stored in the message queue.

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10InfoQueue Interface

ID3D10InfoQueue::GetNumStoredMessagesAllowedByRetrievalFilter method (d3d10sdklayers.h)

Article07/27/2022

Get the number of messages that are able to pass through a retrieval filter.

Syntax

```
C++

UINT64 GetNumStoredMessagesAllowedByRetrievalFilter();
```

Return value

Type: **UINT64**

Number of messages allowed by a retrieval filter.

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10InfoQueue Interface

ID3D10InfoQueue::GetRetrievalFilter method (d3d10sdklayers.h)

Article07/27/2022

Get the retrieval filter at the top of the retrieval-filter stack.

Syntax

C++

```
HRESULT GetRetrievalFilter(
  [out]     D3D10_INFO_QUEUE_FILTER *pFilter,
  [in, out] SIZE_T                  *pFilterByteLength
);
```

Parameters

`[out] pFilter`

Type: **D3D10_INFO_QUEUE_FILTER***

Retrieval filter at the top of the retrieval-filter stack.

`[in, out] pFilterByteLength`

Type: **SIZE_T***

Size of the retrieval filter in bytes. If pFilter is **NULL**, the size of the retrieval filter will be output to this parameter.

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10InfoQueue Interface

ID3D10InfoQueue::GetRetrievalFilterStackSize method (d3d10sdklayers.h)

Article • 07/27/20222 minutes to read

Get the size of the retrieval-filter stack in bytes.

Syntax

```
C++

UINT GetRetrievalFilterStackSize();
```

Return value

Type: **UINT**

Size of the retrieval-filter stack in bytes.

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10InfoQueue Interface

ID3D10InfoQueue::GetStorageFilter method (d3d10sdklayers.h)

Article • 07/27/20222 minutes to read

Get the storage filter at the top of the storage-filter stack.

Syntax

```
C++

HRESULT GetStorageFilter(
  [out]     D3D10_INFO_QUEUE_FILTER *pFilter,
  [in, out] SIZE_T                  *pFilterByteLength
);
```

Parameters

`[out] pFilter`

Type: **D3D10_INFO_QUEUE_FILTER***

Storage filter at the top of the storage-filter stack.

`[in, out] pFilterByteLength`

Type: **SIZE_T***

Size of the storage filter in bytes. If pFilter is **NULL**, the size of the storage filter will be output to this parameter.

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10InfoQueue Interface

ID3D10InfoQueue::GetStorageFilterStackSize method (d3d10sdklayers.h)

Article07/27/2022

Get the size of the storage-filter stack in bytes.

Syntax

```
C++

UINT GetStorageFilterStackSize();
```

Return value

Type: **UINT**

Size of the storage-filter stack in bytes.

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10InfoQueue Interface

ID3D10InfoQueue::PopRetrievalFilter method (d3d10sdklayers.h)

Article • 07/27/20222 minutes to read

Pop a retrieval filter from the top of the retrieval-filter stack.

Syntax

```
C++

void PopRetrievalFilter();
```

Return value

None

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10InfoQueue Interface

ID3D10InfoQueue::PopStorageFilter method (d3d10sdklayers.h)

Article • 07/27/20222 minutes to read

Pop a storage filter from the top of the storage-filter stack.

Syntax

C++

```
void PopStorageFilter();
```

Return value

None

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10InfoQueue Interface

ID3D10InfoQueue::PushCopyOfRetrievalFilter method (d3d10sdklayers.h)

Article • 07/27/20222 minutes to read

Push a copy of retrieval filter currently on the top of the retrieval-filter stack onto the retrieval-filter stack.

Syntax

```
C++

HRESULT PushCopyOfRetrievalFilter();
```

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10InfoQueue Interface

ID3D10InfoQueue::PushCopyOfStorageFilter method (d3d10sdklayers.h)

Article • 07/27/20222 minutes to read

Push a copy of storage filter currently on the top of the storage-filter stack onto the storage-filter stack.

Syntax

```
C++

HRESULT PushCopyOfStorageFilter();
```

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10InfoQueue Interface

ID3D10InfoQueue::PushEmptyRetrievalFilter method (d3d10sdklayers.h)

Article • 07/27/20222 minutes to read

Push an empty retrieval filter onto the retrieval-filter stack.

Syntax

```
C++

HRESULT PushEmptyRetrievalFilter();
```

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

An empty retrieval filter allows all messages to pass through.

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10InfoQueue Interface

ID3D10InfoQueue::PushEmptyStorageFilter method (d3d10sdklayers.h)

Article • 07/27/20222 minutes to read

Push an empty storage filter onto the storage-filter stack.

Syntax

```
C++

HRESULT PushEmptyStorageFilter();
```

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

An empty storage filter allows all messages to pass through.

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10InfoQueue Interface

ID3D10InfoQueue::PushRetrievalFilter method (d3d10sdklayers.h)

Article07/27/2022

Push a retrieval filter onto the retrieval-filter stack.

Syntax

```
C++

HRESULT PushRetrievalFilter(
  [in] D3D10_INFO_QUEUE_FILTER *pFilter
);
```

Parameters

`[in] pFilter`

Type: **D3D10_INFO_QUEUE_FILTER***

Pointer to a retrieval filter (see D3D10_INFO_QUEUE_FILTER).

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10InfoQueue Interface

ID3D10InfoQueue::PushStorageFilter method (d3d10sdklayers.h)

Article • 07/27/20222 minutes to read

Push a storage filter onto the storage-filter stack.

Syntax

```
C++

HRESULT PushStorageFilter(
  [in] D3D10_INFO_QUEUE_FILTER *pFilter
);
```

Parameters

`[in] pFilter`

Type: **D3D10_INFO_QUEUE_FILTER***

Pointer to a storage filter (see D3D10_INFO_QUEUE_FILTER).

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10InfoQueue Interface

ID3D10InfoQueue::SetBreakOnCategory method (d3d10sdklayers.h)

Article • 07/27/20222 minutes to read

Set a message category to break on when a message with that category passes through the storage filter.

Syntax

```
C++

HRESULT SetBreakOnCategory(
  [in] D3D10_MESSAGE_CATEGORY Category,
  [in] BOOL                   bEnable
);
```

Parameters

`[in] Category`

Type: **D3D10_MESSAGE_CATEGORY**

Message category to break on (see D3D10_MESSAGE_CATEGORY).

`[in] bEnable`

Type: **BOOL**

Turns this breaking condition on or off (true for on, false for off).

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10InfoQueue Interface

ID3D10InfoQueue::SetBreakOnID method (d3d10sdklayers.h)

Article • 07/27/20222 minutes to read

Set a message identifier to break on when a message with that identifier passes through the storage filter.

Syntax

```
C++

HRESULT SetBreakOnID(
  [in] D3D10_MESSAGE_ID ID,
  [in] BOOL             bEnable
);
```

Parameters

`[in] ID`

Type: **D3D10_MESSAGE_ID**

Message identifier to break on (see D3D10_MESSAGE_ID).

`[in] bEnable`

Type: **BOOL**

Turns this breaking condition on or off (true for on, false for off).

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10InfoQueue Interface

ID3D10InfoQueue::SetBreakOnSeverity method (d3d10sdklayers.h)

Article • 07/27/20222 minutes to read

Set a message severity level to break on when a message with that severity level passes through the storage filter.

Syntax

```
C++

HRESULT SetBreakOnSeverity(
  [in] D3D10_MESSAGE_SEVERITY Severity,
  [in] BOOL                   bEnable
);
```

Parameters

`[in] Severity`

Type: **D3D10_MESSAGE_SEVERITY**

Message severity level to break on (see D3D10_MESSAGE_SEVERITY).

`[in] bEnable`

Type: **BOOL**

Turns this breaking condition on or off (true for on, false for off).

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10InfoQueue Interface

ID3D10InfoQueue::SetMessageCountLimit method (d3d10sdklayers.h)

Article • 07/27/20222 minutes to read

Set the maximum number of messages that can be added to the message queue.

Syntax

```
C++

HRESULT SetMessageCountLimit(
  [in] UINT64 MessageCountLimit
);
```

Parameters

`[in] MessageCountLimit`

Type: **UINT64**

Maximum number of messages that can be added to the message queue. -1 means no limit.

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

When the number of messages in the message queue has reached the maximum limit, new messages coming in will push old messages out.

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

ID3D10InfoQueue Interface

ID3D10InfoQueue::SetMuteDebugOutput method (d3d10sdklayers.h)

Article • 07/27/20222 minutes to read

Set a boolean that turns the debug output on or off.

Syntax

```
C++

void SetMuteDebugOutput(
  [in] BOOL bMute
);
```

Parameters

`[in] bMute`

Type: **BOOL**

Disable/Enable the debug output (**TRUE** to disable or mute the output, **FALSE** to enable the output).

Return value

None

Remarks

This will stop messages that pass the storage filter from being printed out in the debug output, however those messages will still be added to the message queue.

Requirements

Target Platform	Windows

Header	d3d10sdklayers.h

See also

ID3D10InfoQueue Interface

ID3D10InputLayout interface (d3d10.h)

Article • 02/16/20232 minutes to read

An input-layout interface accesses the input data for the input-assembler stage.

Inheritance

The **ID3D10InputLayout** interface inherits from the ID3D10DeviceChild interface.

Remarks

This interface is created by calling ID3D10Device::CreateInputLayout; use ID3D10Device::IASetInputLayout to bind it to the graphics pipeline.

Requirements

Target Platform	Windows
Header	d3d10.h

See also

Core Interfaces

ID3D10DeviceChild

ID3D10Multithread interface (d3d10.h)

Article • 07/27/20222 minutes to read

A multithread interface accesses multithread settings and can only be used if the thread-safe layer is turned on.

Inheritance

The **ID3D10Multithread** interface inherits from the IUnknown interface. **ID3D10Multithread** also has these types of members:

Methods

The **ID3D10Multithread** interface has these methods.

ID3D10Multithread::Enter Enter a device's critical section. (ID3D10Multithread.Enter)
ID3D10Multithread::GetMultithreadProtected Find out if multithreading is turned on or not.
ID3D10Multithread::Leave Leave a device's critical section. (ID3D10Multithread.Leave)
ID3D10Multithread::SetMultithreadProtected Turn multithreading on or off.

Remarks

This interface is obtained by querying it from the ID3D10Device Interface using IUnknown::QueryInterface.

Requirements

Target Platform	Windows
Header	d3d10.h

See also

Core Interfaces

ID3D10Multithread::Enter method (d3d10.h)

Article07/27/2022

Enter a device's critical section.

Syntax

```C++
void Enter();
```

Return value

None

Remarks

Entering a device's critical section prevents other threads from simultaneously calling that device's methods (if multithread protection is set to true), calling DXGI methods, and calling the methods of all resource, view, shader, state, and asynchronous interfaces.

This function should be used in multithreaded applications when there is a series of graphics commands that must happen in order. This function is typically called at the beginning of the series of graphics commands, and ID3D10Multithread::Leave is typically called after those graphics commands.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Multithread

ID3D10Multithread::GetMultithreadProtected method (d3d10.h)

Article • 06/29/20212 minutes to read

Find out if multithreading is turned on or not.

Syntax

```
C++

BOOL GetMultithreadProtected();
```

Return value

Type: **BOOL**

Whether or not multithreading is turned on. True means on, false means off.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Multithread Interface

ID3D10Multithread::Leave method (d3d10.h)

Article • 07/27/20222 minutes to read

Leave a device's critical section.

Syntax

```
C++

void Leave();
```

Return value

None

Remarks

This function is typically used in multithreaded applications when there is a series of graphics commands that must happen in order. ID3D10Multithread::Enter is typically called at the beginning of a series of graphics commands, and this function is typically called after those graphics commands.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Device Interface

ID3D10Multithread

ID3D10Multithread::SetMultithreadProtected method (d3d10.h)

Article • 10/13/20212 minutes to read

Turn multithreading on or off.

Syntax

```
C++

BOOL SetMultithreadProtected(
  [in] BOOL bMTProtect
);
```

Parameters

`[in] bMTProtect`

Type: **BOOL**

True to turn multithreading on, false to turn it off.

Return value

Type: **BOOL**

True if multithreading was turned on prior to calling this method, false otherwise.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Multithread Interface

ID3D10Predicate interface (d3d10.h)

Article • 02/16/20232 minutes to read

A predicate interface determines whether geometry should be processed depending on the results of a previous draw call.

Inheritance

The **ID3D10Predicate** interface inherits from the ID3D10Query interface.

Remarks

A predicate can be created with ID3D10Device::CreatePredicate, and used with ID3D10Device::SetPredication.

There are two types of predicates in Direct3D 10: stream-output-overflow predicates and occlusion predicates. Stream-output-overflow predicates will cause any geometry residing in stream-output buffers that were overflowed to not be processed. Occlusion predicates will cause any geometry that did not have a single sample pass the depth/stencil tests to not be processed.

For an example of occlusion-predicated rendering, see Draw Predicated Sample.

Requirements

Target Platform	Windows
Header	d3d10.h

See also

Core Interfaces

ID3D10Query

ID3D10Query interface (d3d10.h)

Article • 07/27/20222 minutes to read

A query interface queries information from the GPU.

Inheritance

The **ID3D10Query** interface inherits from ID3D10Asynchronous. **ID3D10Query** also has these types of members:

Methods

The **ID3D10Query** interface has these methods.

ID3D10Query::GetDesc Get a query description. (ID3D10Query.GetDesc)

Remarks

A query can be created with ID3D10Device::CreateQuery.

This interface inherits the functionality of an ID3D10Asynchronous Interface.

Query data is typically gathered by issuing an ID3D10Asynchronous::Begin command, issuing some graphics commands, issuing an ID3D10Asynchronous::End command, and then calling ID3D10Asynchronous::GetData to get data about what happened in between the Begin and End calls. The data returned by GetData will be different depending on the type of query.

There are, however, some queries that do not require calls to Begin. For a list of possible queries see D3D10_QUERY.

A query is typically executed as shown in the following code:

```
D3D10_QUERY_DESC queryDesc;

... // Fill out queryDesc structure
```

```
ID3D10Query * pQuery;
pDevice->CreateQuery(&queryDesc, &pQuery);

pQuery->Begin();

... // Issue graphics commands, do whatever

pQuery->End();

UINT64 queryData; // This data type is different depending on the query type

while( S_OK != pQuery->GetData(&queryData, sizeof(UINT64), 0) )
{
}
```

When using a query that does not require a call to Begin, it still requires a call to End. The call to End causes the data returned by GetData to be accurate up until the last call to End.

Requirements

Target Platform	Windows
Header	d3d10.h

See also

Core Interfaces

ID3D10Asynchronous

ID3D10Query::GetDesc method (d3d10.h)

Article • 07/27/20222 minutes to read

Get a query description.

Syntax

C++

```
void GetDesc(
  [out] D3D10_QUERY_DESC *pDesc
);
```

Parameters

`[out] pDesc`

Type: **D3D10_QUERY_DESC***

Pointer to a query description (see D3D10_QUERY_DESC).

Return value

None

Requirements

Target Platform	Windows
Header	d3d10.h (include D3d10core)

See also

ID3D10Query Interface

ID3D10RasterizerState interface (d3d10.h)

Article • 07/22/20212 minutes to read

A rasterizer-state interface accesses rasterizer state for the rasterizer stage.

Inheritance

The **ID3D10RasterizerState** interface inherits from ID3D10DeviceChild. **ID3D10RasterizerState** also has these types of members:

Methods

The **ID3D10RasterizerState** interface has these methods.

ID3D10RasterizerState::GetDesc Get the properties of a rasterizer-state object.

Remarks

A rasterizer-state object is created with ID3D10Device::CreateRasterizerState and bound to the pipeline with ID3D10Device::RSSetState.

Requirements

Target Platform	Windows
Header	d3d10.h

See also

Core Interfaces

ID3D10DeviceChild

ID3D10RasterizerState::GetDesc method (d3d10.h)

Article • 10/13/20212 minutes to read

Get the properties of a rasterizer-state object.

Syntax

```
C++

void GetDesc(
  [out] D3D10_RASTERIZER_DESC *pDesc
);
```

Parameters

`[out] pDesc`

Type: **D3D10_RASTERIZER_DESC***

Pointer to a rasterizer-state description (see D3D10_RASTERIZER_DESC).

Return value

None

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10RasterizerState Interface

ID3D10SamplerState interface (d3d10.h)

Article07/22/2021

A sampler-state interface accesses sampler state for a texture.

Inheritance

The **ID3D10SamplerState** interface inherits from ID3D10DeviceChild.
ID3D10SamplerState also has these types of members:

Methods

The **ID3D10SamplerState** interface has these methods.

ID3D10SamplerState::GetDesc Get the sampler state.

Remarks

Create a sampler-state object by calling ID3D10Device::CreateSamplerState.

To initialize sampler state, bind the sampler-state object to the pipeline by calling ID3D10Device::VSSetSamplers, ID3D10Device::GSSetSamplers, or ID3D10Device::PSSetSamplers.

Requirements

Target Platform	Windows
Header	d3d10.h

See also

Core Interfaces

Core Reference

ID3D10DeviceChild

ID3D10SamplerState::GetDesc method (d3d10.h)

Article10/13/2021

Get the sampler state.

Syntax

C++

```
void GetDesc(
  [out] D3D10_SAMPLER_DESC *pDesc
);
```

Parameters

`[out] pDesc`

Type: **D3D10_SAMPLER_DESC***

A pointer to the sampler state (see D3D10_SAMPLER_DESC).

Return value

None

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10SamplerState Interface

ID3D10StateBlock interface (d3d10effect.h)

Article • 07/27/20222 minutes to read

A state-block interface encapsulates render states.

Inheritance

The **ID3D10StateBlock** interface inherits from the IUnknown interface. **ID3D10StateBlock** also has these types of members:

Methods

The **ID3D10StateBlock** interface has these methods.

ID3D10StateBlock::Apply Apply the state block to the current device state. (ID3D10StateBlock.Apply)
ID3D10StateBlock::Capture Capture the current value of states that are included in a stateblock. (ID3D10StateBlock.Capture)
ID3D10StateBlock::GetDevice Get the device.
ID3D10StateBlock::ReleaseAllDeviceObjects Release all references to device objects.

Remarks

To create a state-block interface, call D3D10CreateStateBlock.

This interface can be used to save and restore pipeline state. It can also be used to capture the current state.

Requirements

Target Platform	Windows
Header	d3d10effect.h (include D3D10.h)

See also

Core Interfaces

ID3D10StateBlock::Apply method (d3d10effect.h)

Article07/27/2022

Apply the state block to the current device state.

Syntax

```
C++

HRESULT Apply();
```

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h (include D3D10.h)
Library	D3D10.lib

See also

ID3D10StateBlock Interface

ID3D10StateBlock::Capture method (d3d10effect.h)

Article • 07/27/20222 minutes to read

Capture the current value of states that are included in a stateblock.

Syntax

```
C++

HRESULT Capture();
```

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Remarks

Capture captures current values for states within an existing state block. It does not capture the entire state of the device. Creating an empty stateblock and calling Capture does nothing if no states have been set.

Requirements

Target Platform	Windows
Header	d3d10effect.h (include D3D10.h)
Library	D3D10.lib

See also

ID3D10StateBlock Interface

ID3D10StateBlock::GetDevice method (d3d10effect.h)

Article10/13/2021

Get the device.

Syntax

```
C++

HRESULT GetDevice(
  [out] ID3D10Device **ppDevice
);
```

Parameters

`[out] ppDevice`

Type: **ID3D10Device****

Pointer to the ID3D10Device interface that is returned.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h (include D3D10.h)
Library	D3D10.lib

See also

ID3D10StateBlock Interface

ID3D10StateBlock::ReleaseAllDeviceObjects method (d3d10effect.h)

Article • 06/29/20212 minutes to read

Release all references to device objects.

Syntax

```
C++

HRESULT ReleaseAllDeviceObjects();
```

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Remarks

Each time you return a pointer to an interface (by calling ID3D10StateBlock::GetDevice), the internal reference count is incremented; when you are finished using a stateblock, call this method to release all references and avoid a memory leak.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10StateBlock Interface

ID3D10SwitchToRef interface (d3d10sdklayers.h)

Article • 07/22/20212 minutes to read

A switch-to-reference interface (see the switch-to-reference layer) enables an application to switch between a hardware and software device.

Inheritance

The **ID3D10SwitchToRef** interface inherits from ID3D10DeviceChild. **ID3D10SwitchToRef** also has these types of members:

Methods

The **ID3D10SwitchToRef** interface has these methods.

ID3D10SwitchToRef::GetUseRef Get a boolean value that indicates the type of device being used.
ID3D10SwitchToRef::SetUseRef Switch between a hardware and a software device.

Remarks

This interface is obtained by calling QueryInterface on a ID3D10Device Interface created with the D3D10_CREATE_DEVICE_SWITCH_TO_REF flag.

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

Core Interfaces

ID3D10DeviceChild

ID3D10SwitchToRef::GetUseRef method (d3d10sdklayers.h)

Article • 06/29/20212 minutes to read

Get a boolean value that indicates the type of device being used.

Syntax

```
C++

BOOL GetUseRef();
```

Return value

Type: **BOOL**

TRUE if the device is a software device, **FALSE** if the device is a hardware device. See remarks.

Remarks

A hardware device is commonly referred to as a HAL device, which stands for a hardware accelerated device. This means that the pipeline is rendering all of the pipeline commands in hardware, using the GPU. Operating the pipeline with a HAL device gives the best performance generally, but it can be more difficult to debug since resources exist on the GPU instead of the CPU.

A software device implements rendering in software using the CPU with no hardware acceleration. A software device is commonly referred to as a reference device or REF device. Because a REF device implements rendering on the CPU, it is generally slower, but is easier to debug since it allows access to resources.

Requirements

Target Platform	Windows

Header	d3d10sdklayers.h

See also

ID3D10SwitchToRef Interface

ID3D10SwitchToRef::SetUseRef method (d3d10sdklayers.h)

Article • 10/13/20212 minutes to read

Switch between a hardware and a software device.

Syntax

```
C++

BOOL SetUseRef(
  [in] BOOL UseRef
);
```

Parameters

`[in] UseRef`

Type: **BOOL**

A boolean value. Set this to **TRUE** to change to a software device, set this to **FALSE** to change to a hardware device.

Return value

Type: **BOOL**

The previous value of *UseRef*.

Remarks

This API will fail if the device is not switchable; you must have created a device that is switchable by specifying the D3D10_CREATE_DEVICE_SWITCH_TO_REF flag during device creation (when calling D3D10CreateDevice).

Switching from a software device to a hardware device clears all cached objects from system memory. Switching from a hardware device to a software device causes resources to be downloaded to system memory.

Requirements

Target Platform	Windows
Header	d3d10sdklayers.h

See also

D3D10CreateDevice

ID3D10SwitchToRef Interface

Direct3D 10 core functions

Article • 01/06/2021 • 2 minutes to read

This section contains information about the following core functions:

Functions	Description
D3D10CreateBlob	Create a buffer.
D3D10CreateDevice	Create a Direct3D 10 device.
D3D10CreateDevice1	Create a Direct3D 10.1 device.
D3D10CreateDeviceAndSwapChain	Create a Direct3D 10 device and a swap chain.
D3D10CreateDeviceAndSwapChain1	Create a Direct3D 10.1 device and a swap chain.

Related topics

Core Reference

D3D10CreateBlob function (d3d10misc.h)

Article10/13/2021

Create a buffer.

> **Note** Instead of using this function, we recommend that you use the **D3DCreateBlob** API.

Syntax

C++

```
HRESULT D3D10CreateBlob(
  [in]  SIZE_T      NumBytes,
  [out] LPD3D10BLOB *ppBuffer
);
```

Parameters

`[in] NumBytes`

Type: **SIZE_T**

Number of bytes in the blob.

`[out] ppBuffer`

Type: **LPD3D10BLOB** *

The address of a pointer to the buffer (see ID3D10Blob Interface).

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10misc.h
Library	D3D10.lib
DLL	D3D10.dll

See also

Core Functions

D3D10CreateDevice function (d3d10misc.h)

Article • 10/13/20212 minutes to read

Create a Direct3D 10.0 device that represents the display adapter.

Syntax

C++

```
HRESULT D3D10CreateDevice(
  [in]  IDXGIAdapter      *pAdapter,
  [in]  D3D10_DRIVER_TYPE DriverType,
  [in]  HMODULE           Software,
  [in]  UINT              Flags,
  [in]  UINT              SDKVersion,
  [out] ID3D10Device      **ppDevice
);
```

Parameters

`[in] pAdapter`

Type: **IDXGIAdapter***

Pointer to the display adapter (see IDXGIAdapter) when creating a hardware device; otherwise set this parameter to **NULL**. If **NULL** is specified when creating a hardware device, Direct3D will use the first adapter enumerated by EnumAdapters.

`[in] DriverType`

Type: **D3D10_DRIVER_TYPE**

The device-driver type (see D3D10_DRIVER_TYPE). The driver type determines the type of device you will create.

`[in] Software`

Type: **HMODULE**

Reserved. Set to **NULL**.

`[in] Flags`

Type: **UINT**

Optional. Device creation flags (see D3D10_CREATE_DEVICE_FLAG) that enable API layers. These flags can be bitwise OR'd together.

`[in] SDKVersion`

Type: **UINT**

Bit flag that indicates the version of the SDK. Should always be D3D10_SDK_VERSION.

`[out] ppDevice`

Type: **ID3D10Device****

Address of a pointer to the device created (see ID3D10Device Interface).

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

This example creates a reference device.

```
ID3D10Device* g_pd3dDevice = NULL;
D3D10CreateDevice( NULL, D3D10_DRIVER_TYPE_REFERENCE, NULL, 0,
    D3D10_SDK_VERSION, &g_pd3dDevice );
```

To create a device and a swap chain at the same time, see D3D10CreateDeviceAndSwapChain.

The object returned by D3D10CreateDevice implements the IUnknown interface and can be queried for other interfaces the object supports. To retrieve the IDXGIDevice interface of the object the following code could be used.

```
IDXGIDevice * pDXGIDevice;
hr = g_pd3dDevice->QueryInterface(__uuidof(IDXGIDevice), (void
**)&pDXGIDevice);
```

Requirements

Target Platform	Windows
Header	d3d10misc.h
Library	D3D10.lib
DLL	D3D10.dll

See also

Core Functions

D3D10CreateDevice1 function (d3d10_1.h)

Article • 10/13/20212 minutes to read

Create a Direct3D 10.1 device that represents the display adapter.

Syntax

C++

```
HRESULT D3D10CreateDevice1(
  [in]  IDXGIAdapter         *pAdapter,
  [in]  D3D10_DRIVER_TYPE    DriverType,
  [in]  HMODULE              Software,
  [in]  UINT                 Flags,
  [in]  D3D10_FEATURE_LEVEL1 HardwareLevel,
  [in]  UINT                 SDKVersion,
  [out] ID3D10Device1        **ppDevice
);
```

Parameters

`[in] pAdapter`

Type: **IDXGIAdapter***

Pointer to the display adapter (see IDXGIAdapter) when creating a hardware device; otherwise set this parameter to **NULL**. If **NULL** is specified when creating a hardware device, Direct3D will use the first adapter enumerated by EnumAdapters.

`[in] DriverType`

Type: **D3D10_DRIVER_TYPE**

The device-driver type (see D3D10_DRIVER_TYPE). The driver type determines the type of device you will create.

`[in] Software`

Type: **HMODULE**

This is set to **NULL** except for D3D10_DRIVER_TYPE_SOFTWARE driver types.

`[in] Flags`

Type: **UINT**

Optional. Device creation flags (see D3D10_CREATE_DEVICE_FLAG) that enable API layers. These flags can be bitwise OR'd together.

`[in] HardwareLevel`

Type: **D3D10_FEATURE_LEVEL1**

The version of hardware that is available for acceleration (see D3D10_FEATURE_LEVEL1).

`[in] SDKVersion`

Type: **UINT**

Bit flag that indicates the version of the SDK. Should be D3D10_1_SDK_VERSION, defined in D3D10.h.

`[out] ppDevice`

Type: **ID3D10Device1****

Address of a pointer to the device created (see ID3D10Device1 Interface).

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

To create a device and a swap chain at the same time, see D3D10CreateDeviceAndSwapChain1.

This method requires Windows Vista Service Pack 1, Windows Server 2008, or later release of Windows.

The object returned by D3D10CreateDevice1 implements the IUnknown interface and can be queried for other interfaces the object supports. To retrieve the IDXGIDevice interface of the object the following code could be used.

```
IDXGIDevice * pDXGIDevice;
hr = g_pd3dDevice->QueryInterface(__uuidof(IDXGIDevice), (void
**)&pDXGIDevice);
```

Requirements

Target Platform	Windows
Header	d3d10_1.h
Library	D3D10_1.lib
DLL	D3D10_1.dll

See also

Core Functions

D3D10CreateDeviceAndSwapChain function (d3d10misc.h)

Article • 10/13/20212 minutes to read

Create a Direct3D 10.0 device and a swap chain.

Syntax

C++

```
HRESULT D3D10CreateDeviceAndSwapChain(
  [in]  IDXGIAdapter         *pAdapter,
  [in]  D3D10_DRIVER_TYPE    DriverType,
  [in]  HMODULE              Software,
  [in]  UINT                 Flags,
  [in]  UINT                 SDKVersion,
  [in]  DXGI_SWAP_CHAIN_DESC *pSwapChainDesc,
  [out] IDXGISwapChain       **ppSwapChain,
  [out] ID3D10Device         **ppDevice
);
```

Parameters

`[in] pAdapter`

Type: **IDXGIAdapter***

Pointer to a IDXGIAdapter.

`[in] DriverType`

Type: **D3D10_DRIVER_TYPE**

The type of driver for the device. See D3D10_DRIVER_TYPE.

`[in] Software`

Type: **HMODULE**

A handle to the DLL that implements a software rasterizer. Must be **NULL** if DriverType is non-software. The HMODULE of a DLL can be obtained with LoadLibrary, LoadLibraryEx, or GetModuleHandle.

`[in] Flags`

Type: **UINT**

Optional. Device creation flags (see D3D10_CREATE_DEVICE_FLAG) that enable API layers. These flags can be bitwise OR'd together.

`[in] SDKVersion`

Type: **UINT**

Bit flag that indicates the version of the SDK. Should be D3D10_SDK_VERSION, defined in d3d10.h.

`[in] pSwapChainDesc`

Type: **DXGI_SWAP_CHAIN_DESC***

Description of the swap chain. See DXGI_SWAP_CHAIN_DESC.

`[out] ppSwapChain`

Type: **IDXGISwapChain****

Address of a pointer to an IDXGISwapChain.

`[out] ppDevice`

Type: **ID3D10Device****

Address of a pointer to an ID3D10Device Interface that will receive the newly created device.

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

To create a device without creating a swap chain, see D3D10CreateDevice.

> **Note** If you call this API in a Session 0 process, it returns **DXGI_ERROR_NOT_CURRENTLY_AVAILABLE**.

Requirements

Target Platform	Windows
Header	d3d10misc.h
Library	D3D10.lib
DLL	D3D10.dll

See also

Core Functions

D3D10CreateDeviceAndSwapChain1 function (d3d10_1.h)

Article • 10/13/20212 minutes to read

Create a Direct3D 10.1 device and a swap chain.

Syntax

C++

```
HRESULT D3D10CreateDeviceAndSwapChain1(
  [in]  IDXGIAdapter         *pAdapter,
  [in]  D3D10_DRIVER_TYPE    DriverType,
  [in]  HMODULE              Software,
  [in]  UINT                 Flags,
  [in]  D3D10_FEATURE_LEVEL1 HardwareLevel,
  [in]  UINT                 SDKVersion,
  [in]  DXGI_SWAP_CHAIN_DESC *pSwapChainDesc,
  [out] IDXGISwapChain       **ppSwapChain,
  [out] ID3D10Device1        **ppDevice
);
```

Parameters

`[in] pAdapter`

Type: **IDXGIAdapter***

Pointer to a IDXGIAdapter.

`[in] DriverType`

Type: **D3D10_DRIVER_TYPE**

The type of driver for the device. See D3D10_DRIVER_TYPE.

`[in] Software`

Type: **HMODULE**

A handle to the DLL that implements a software rasterizer. Must be **NULL** if DriverType is non-software. The HMODULE of a DLL can be obtained with LoadLibrary, LoadLibraryEx, or GetModuleHandle.

`[in] Flags`

Type: **UINT**

Optional. Device creation flags (see D3D10_CREATE_DEVICE_FLAG) that enable API layers. These flags can be bitwise OR'd together.

`[in] HardwareLevel`

Type: **D3D10_FEATURE_LEVEL1**

The version of hardware that is available for acceleration (see D3D10_FEATURE_LEVEL1).

`[in] SDKVersion`

Type: **UINT**

Bit flag that indicates the version of the SDK. Should be D3D10_1_SDK_VERSION, defined in D3D10.h.

`[in] pSwapChainDesc`

Type: **DXGI_SWAP_CHAIN_DESC***

Description of the swap chain. See DXGI_SWAP_CHAIN_DESC.

`[out] ppSwapChain`

Type: **IDXGISwapChain****

Address of a pointer to an IDXGISwapChain.

`[out] ppDevice`

Type: **ID3D10Device1****

Address of a pointer to an ID3D10Device1 Interface that will receive the newly created device.

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

To create a device without creating a swap chain, see D3D10CreateDevice1.

This method requires Windows Vista Service Pack 1, Windows Server 2008, or later release of Windows.

> **Note** If you call this API in a Session 0 process, it returns **DXGI_ERROR_NOT_CURRENTLY_AVAILABLE**.

Requirements

Target Platform	Windows
Header	d3d10_1.h
Library	D3D10_1.lib
DLL	D3D10_1.dll

See also

Core Functions

Direct3D 10 core structures

Article • 01/06/2021 • 2 minutes to read

This section contains information about the following core structures:

Structures	Description
D3D10_BLEND_DESC	Blending options for a Direct3D 10.0 device.
D3D10_BLEND_DESC1	Blending options for a Direct3D 10.1 device.
D3D10_BOX	A rectangular region, often in a texture.
D3D10_COUNTER_DESC	A performance-counter description.
D3D10_COUNTER_INFO	Information about the video card's performance counter capabilities.
D3D10_DEPTH_STENCIL_DESC	A depth-stencil description.
D3D10_DEPTH_STENCILOP_DESC	A depth-stencil operation.
D3D10_INFO_QUEUE_FILTER	A debug-message filter.
D3D10_INFO_QUEUE_FILTER_DESC	A debug-message-filter description.
D3D10_INPUT_ELEMENT_DESC	A vertex-buffer element description for an input slot.
D3D10_QUERY_DATA_PIPELINE_STATISTICS	Pipeline query statistics.
D3D10_QUERY_DATA_SO_STATISTICS	Stream-output stage query statistics.
D3D10_QUERY_DATA_TIMESTAMP_DISJOINT	Timestamp query statistics.
D3D10_QUERY_DESC	A query description.
D3D10_MESSAGE	A debug message for the information queue.
D3D10_RASTERIZER_DESC	Rasterizer-stage description.
D3D10_RECT	A 2D rectangle.
D3D10_RENDER_TARGET_BLEND_DESC1	Blending options for a Direct3D 10.1 render target.
D3D10_SAMPLER_DESC	A sampler description.
D3D10_SHADER_RESOURCE_VIEW_DESC	Describes a shader-resource view for Direct3D 10.0.

Structures	Description
D3D10_SHADER_RESOURCE_VIEW_DESC1	Describes a shader-resource view for Direct3D 10.1.
D3D10_SIGNATURE_PARAMETER_DESC	A shader-parameter description.
D3D10_STATE_BLOCK_MASK	A state-block mask.
D3D10_SO_DECLARATION_ENTRY	A vertex-buffer element description for an output slot.
D3D10_VIEWPORT	Viewport dimensions.

In addition, there is a 2D rectangle structure defined in d3d10.h.

```
typedef RECT D3D10_RECT;
```

For documentation, see RECT.

Related topics

Core Reference

D3D10_BLEND_DESC structure (d3d10.h)

Article • 09/01/20222 minutes to read

Describes the blend state.

Syntax

C++

```
typedef struct D3D10_BLEND_DESC {
  BOOL           AlphaToCoverageEnable;
  BOOL           BlendEnable[8];
  D3D10_BLEND    SrcBlend;
  D3D10_BLEND    DestBlend;
  D3D10_BLEND_OP BlendOp;
  D3D10_BLEND    SrcBlendAlpha;
  D3D10_BLEND    DestBlendAlpha;
  D3D10_BLEND_OP BlendOpAlpha;
  UINT8          RenderTargetWriteMask[8];
} D3D10_BLEND_DESC;
```

Members

`AlphaToCoverageEnable`

Type: **BOOL**

Determines whether or not to use alpha-to-coverage as a multisampling technique when setting a pixel to a rendertarget.

`BlendEnable[8]`

Type: **BOOL**

Enable (or disable) blending. There are eight elements in this array; these correspond to the eight rendertargets that can be set to output-merger stage at one time.

`SrcBlend`

Type: **D3D10_BLEND**

This blend option specifies the first RGB data source and includes an optional pre-blend operation.

`DestBlend`

Type: **D3D10_BLEND**

This blend option specifies the second RGB data source and includes an optional pre-blend operation.

`BlendOp`

Type: **D3D10_BLEND_OP**

This blend operation defines how to combine the RGB data sources.

`SrcBlendAlpha`

Type: **D3D10_BLEND**

This blend option specifies the first alpha data source and includes an optional pre-blend operation. Blend options that end in _COLOR are not allowed.

`DestBlendAlpha`

Type: **D3D10_BLEND**

This blend option specifies the second alpha data source and includes an optional pre-blend operation. Blend options that end in _COLOR are not allowed.

`BlendOpAlpha`

Type: **D3D10_BLEND_OP**

This blend operation defines how to combine the alpha data sources.

`RenderTargetWriteMask[8]`

Type: **UINT8**

A per-pixel write mask that allows control over which components can be written (see D3D10_COLOR_WRITE_ENABLE).

Remarks

To see how blending is done, see Output-Merger Stage (Direct3D 10).

These are the default values for blend state.

State	Default Value

AlphaToCoverageEnable	FALSE
BlendEnable[8]	FALSE (for all 8)
SrcBlend	D3D10_BLEND_ONE
DestBlend	D3D10_BLEND_ZERO
BlendOp	D3D10_BLEND_OP_ADD
SrcBlendAlpha	D3D10_BLEND_ONE
DestBlendAlpha	D3D10_BLEND_ZERO
BlendOpAlpha	D3D10_BLEND_OP_ADD
RenderTargetWriteMask[8]	D3D10_COLOR_WRITE_ENABLE_ALL (for all 8)

Requirements

Header	d3d10.h

See also

Core Structures

D3D10_BLEND_DESC1 structure (d3d10_1.h)

Article • 09/01/20222 minutes to read

Describes the blend state for a Direct3D 10.1 device.

Syntax

C++

```
typedef struct D3D10_BLEND_DESC1 {
  BOOL                            AlphaToCoverageEnable;
  BOOL                            IndependentBlendEnable;
  D3D10_RENDER_TARGET_BLEND_DESC1 RenderTarget[8];
} D3D10_BLEND_DESC1;
```

Members

`AlphaToCoverageEnable`

Type: **BOOL**

Determines whether or not to use the alpha-to-coverage multisampling technique when setting a render-target pixel.

`IndependentBlendEnable`

Type: **BOOL**

Set to **TRUE** to enable independent blending in simultaneous render targets. If set to **FALSE**, only the RenderTarget[0] members are used. RenderTarget[1..7] are ignored.

`RenderTarget[8]`

Type: **D3D10_RENDER_TARGET_BLEND_DESC1**

An array of render-target-blend descriptions (see D3D10_RENDER_TARGET_BLEND_DESC1); these correspond to the eight rendertargets that can be set to the output-merger stage at one time.

Remarks

To see how blending is done, see Output-Merger Stage (Direct3D 10).

These are the default values for the blend description.

State	Default Value
AlphaToCoverageEnable	**FALSE**
IndependentBlendEnable	**FALSE**
RenderTarget[0].BlendEnable	**FALSE**
RenderTarget[0].SrcBlend	D3D10_BLEND_ONE
RenderTarget[0].DestBlend	D3D10_BLEND_ZERO
RenderTarget[0].BlendOp	D3D10_BLEND_OP_ADD
RenderTarget[0].SrcBlendAlpha	D3D10_BLEND_ONE
RenderTarget[0].DestBlendAlpha	D3D10_BLEND_ZERO
RenderTarget[0].BlendOpAlpha	D3D10_BLEND_OP_ADD
RenderTarget[0].RenderTargetWriteMask	D3D10_COLOR_WRITE_ENABLE_ALL

This structure requires Windows Vista Service Pack 1.

If the driver type is set to D3D_DRIVER_TYPE_HARDWARE, the feature level is set to less than or equal to D3D_FEATURE_LEVEL_9_3, and the pixel format of the render target is set to DXGI_FORMAT_R8G8B8A8_UNORM_SRGB, **DXGI_FORMAT_B8G8R8A8_UNORM_SRGB**, or **DXGI_FORMAT_B8G8R8X8_UNORM_SRGB**, the device performs the blend in standard RGB (sRGB) space and not in linear space. However, if the feature level is set to greater than **D3D_FEATURE_LEVEL_9_3**, the device performs the blend in linear space.

Requirements

Header	d3d10_1.h

See also

Core Structures

D3D10_BOX structure (d3d10.h)

Article07/27/2022

Defines a 3D box.

Syntax

C++

```
typedef struct D3D10_BOX {
  UINT left;
  UINT top;
  UINT front;
  UINT right;
  UINT bottom;
  UINT back;
} D3D10_BOX;
```

Members

`left`

Type: **UINT**

The x position of the left hand side of the box.

`top`

Type: **UINT**

The y position of the top of the box.

`front`

Type: **UINT**

The z position of the front of the box.

`right`

Type: **UINT**

The x position of the right hand side of the box.

`bottom`

Type: **UINT**

The y position of the bottom of the box.

`back`

Type: **UINT**

The z position of the back of the box.

Remarks

The following diagram shows a 3D box, where the origin is the left, front, top corner.

Requirements

Header	d3d10.h

See also

Core Structures

D3D10_COUNTER_DESC structure (d3d10.h)

Article07/27/2022

Describes a counter.

Syntax

```
C++

typedef struct D3D10_COUNTER_DESC {
  D3D10_COUNTER Counter;
  UINT          MiscFlags;
} D3D10_COUNTER_DESC;
```

Members

`Counter`

Type: **D3D10_COUNTER**

Type of counter (see D3D10_COUNTER).

`MiscFlags`

Type: **UINT**

Reserved.

Requirements

Header	d3d10.h

See also

Core Structures

D3D10_COUNTER_INFO structure (d3d10.h)

Article • 07/27/20222 minutes to read

Information about the video card's performance counter capabilities.

Syntax

C++

```
typedef struct D3D10_COUNTER_INFO {
  D3D10_COUNTER LastDeviceDependentCounter;
  UINT          NumSimultaneousCounters;
  UINT8         NumDetectableParallelUnits;
} D3D10_COUNTER_INFO;
```

Members

`LastDeviceDependentCounter`

Type: **D3D10_COUNTER**

Largest device-dependent counter ID that the device supports. If none are supported, this value will be 0. Otherwise it will be greater than or equal to D3D10_COUNTER_DEVICE_DEPENDENT_0. See D3D10_COUNTER.

`NumSimultaneousCounters`

Type: **UINT**

Number of counters that can be simultaneously supported.

`NumDetectableParallelUnits`

Type: **UINT8**

Number of detectable parallel units that the counter is able to discern. Values are 1 ~ 4. Use NumDetectableParallelUnits to interpret the values of the VERTEX_PROCESSING, GEOMETRY_PROCESSING, PIXEL_PROCESSING, and OTHER_GPU_PROCESSING counters. See ID3D10Asynchronous::GetData for an equation.

Remarks

This structure is returned by ID3D10Device::CheckCounterInfo.

Requirements

Header	d3d10.h

See also

Core Structures

D3D10_DEPTH_STENCILOP_DESC structure (d3d10.h)

Article • 04/02/20212 minutes to read

Describes the stencil operations that can be performed based on the results of stencil test.

Syntax

C++

```
typedef struct D3D10_DEPTH_STENCILOP_DESC {
  D3D10_STENCIL_OP      StencilFailOp;
  D3D10_STENCIL_OP      StencilDepthFailOp;
  D3D10_STENCIL_OP      StencilPassOp;
  D3D10_COMPARISON_FUNC StencilFunc;
} D3D10_DEPTH_STENCILOP_DESC;
```

Members

`StencilFailOp`

Type: **D3D10_STENCIL_OP**

A member of the D3D10_STENCIL_OP enumerated type that describes the stencil operation to perform when stencil testing fails. The default value is **D3D10_STENCIL_OP_KEEP**.

`StencilDepthFailOp`

Type: **D3D10_STENCIL_OP**

A member of the D3D10_STENCIL_OP enumerated type that describes the stencil operation to perform when stencil testing passes and depth testing fails. The default value is **D3D10_STENCIL_OP_KEEP**.

`StencilPassOp`

Type: **D3D10_STENCIL_OP**

A member of the D3D10_STENCIL_OP enumerated type that describes the stencil operation to perform when stencil testing and depth testing both pass. The default

value is **D3D10_STENCIL_OP_KEEP**.

`StencilFunc`

Type: **D3D10_COMPARISON_FUNC**

A member of the D3D10_COMPARISON_FUNC enumerated type that describes how stencil data is compared against existing stencil data. The default value is **D3D10_COMPARISON_ALWAYS**.

Remarks

The stencil operation can be set differently based on the outcome of the stencil test by using the **StencilFunc** member. This can be done for the stencil test portion of depth-stencil testing.

The D3D10_DEPTH_STENCILOP_DESC structure is a member of the D3D10_DEPTH_STENCIL_DESC structure.

Requirements

Header	d3d10.h

See also

Core Structures

D3D10_DEPTH_STENCIL_DESC structure (d3d10.h)

Article • 07/27/20222 minutes to read

Describes depth-stencil state.

Syntax

C++

```
typedef struct D3D10_DEPTH_STENCIL_DESC {
  BOOL                        DepthEnable;
  D3D10_DEPTH_WRITE_MASK      DepthWriteMask;
  D3D10_COMPARISON_FUNC       DepthFunc;
  BOOL                        StencilEnable;
  UINT8                       StencilReadMask;
  UINT8                       StencilWriteMask;
  D3D10_DEPTH_STENCILOP_DESC FrontFace;
  D3D10_DEPTH_STENCILOP_DESC BackFace;
} D3D10_DEPTH_STENCIL_DESC;
```

Members

`DepthEnable`

Type: **BOOL**

A Boolean value that enables depth testing. The default value is **TRUE**.

`DepthWriteMask`

Type: **D3D10_DEPTH_WRITE_MASK**

A member of the D3D10_DEPTH_WRITE_MASK enumerated type that identifies a portion of the depth-stencil buffer that can be modified by depth data. The default value is **D3D10_DEPTH_WRITE_MASK_ALL**.

`DepthFunc`

Type: **D3D10_COMPARISON_FUNC**

A member of the D3D10_COMPARISON_FUNC enumerated type that defines how depth data is compared against existing depth data. The default value is

D3D10_COMPARISON_LESS

`StencilEnable`

Type: **BOOL**

A Boolean value that enables stencil testing. The default value is **FALSE**.

`StencilReadMask`

Type: **UINT8**

A value that identifies a portion of the depth-stencil buffer for reading stencil data. The default value is **D3D10_DEFAULT_STENCIL_READ_MASK**.

`StencilWriteMask`

Type: **UINT8**

A value that identifies a portion of the depth-stencil buffer for writing stencil data. The default value is **D3D10_DEFAULT_STENCIL_WRITE_MASK**.

`FrontFace`

Type: **D3D10_DEPTH_STENCILOP_DESC**

A D3D10_DEPTH_STENCILOP_DESC structure that identifies how to use the results of the depth test and the stencil test for pixels whose surface normal is facing toward the camera.

`BackFace`

Type: **D3D10_DEPTH_STENCILOP_DESC**

A D3D10_DEPTH_STENCILOP_DESC structure that identifies how to use the results of the depth test and the stencil test for pixels whose surface normal is facing away from the camera.

Remarks

Depth-stencil state controls how depth-stencil testing is performed by the output-merger stage.

The formats that support stenciling are DXGI_FORMAT_D24_UNORM_S8_UINT and DXGI_FORMAT_D32_FLOAT_S8X24_UINT.

Requirements

Header	d3d10.h

See also

Core Structures

D3D10_INFO_QUEUE_FILTER structure (d3d10sdklayers.h)

Article07/27/2022

Debug message filter; contains a lists of message types to allow or deny.

Syntax

C++

```
typedef struct D3D10_INFO_QUEUE_FILTER {
  D3D10_INFO_QUEUE_FILTER_DESC AllowList;
  D3D10_INFO_QUEUE_FILTER_DESC DenyList;
} D3D10_INFO_QUEUE_FILTER;
```

Members

`AllowList`

Type: **D3D10_INFO_QUEUE_FILTER_DESC**

A D3D10_INFO_QUEUE_FILTER_DESC structure describing the types of messages the info queue should allow.

`DenyList`

Type: **D3D10_INFO_QUEUE_FILTER_DESC**

A D3D10_INFO_QUEUE_FILTER_DESC structure describing the types of messages the info queue should reject.

Remarks

For use with an ID3D10InfoQueue Interface.

Providing an allow list with non-zero values causes only the specified combination of categories, severities and message IDs to be allowed.
Messages that do not match the specified combination will be rejected.

Providing a deny list with non-zero values causes the specified combination of categories, severities and message IDs to be rejected. Messages that do not match the

specified combination will be allowed.

Requirements

Header	d3d10sdklayers.h (include D3D10.h)

See also

Core Structures

D3D10_INFO_QUEUE_FILTER_DESC structure (d3d10sdklayers.h)

Article07/27/2022

Allow or deny certain types of messages to pass through a filter.

Syntax

C++

```
typedef struct D3D10_INFO_QUEUE_FILTER_DESC {
  UINT                   NumCategories;
  D3D10_MESSAGE_CATEGORY *pCategoryList;
  UINT                   NumSeverities;
  D3D10_MESSAGE_SEVERITY *pSeverityList;
  UINT                   NumIDs;
  D3D10_MESSAGE_ID       *pIDList;
} D3D10_INFO_QUEUE_FILTER_DESC;
```

Members

`NumCategories`

Type: **UINT**

Number of message categories to allow or deny.

`pCategoryList`

Type: **D3D10_MESSAGE_CATEGORY***

Array of message categories to allow or deny. Array must have at least NumCategories members (see D3D10_MESSAGE_CATEGORY).

`NumSeverities`

Type: **UINT**

Number of message severity levels to allow or deny.

`pSeverityList`

Type: **D3D10_MESSAGE_SEVERITY***

Array of message severity levels to allow or deny. Array must have at least NumSeverities members (see D3D10_MESSAGE_SEVERITY).

`NumIDs`

Type: **UINT**

Number of message IDs to allow or deny.

`pIDList`

Type: **D3D10_MESSAGE_ID***

Array of message IDs to allow or deny. Array must have at least NumIDs members (see D3D10_MESSAGE_ID).

Remarks

D3D10_INFO_QUEUE_FILTER_DESC is used to define the allow list and deny list in the D3D10_INFO_QUEUE_FILTER structure.

Requirements

Header	d3d10sdklayers.h (include D3D10.h)

See also

Core Structures

D3D10_INPUT_ELEMENT_DESC structure (d3d10.h)

Article • 07/27/20222 minutes to read

A description of a single element for the input-assembler stage.

Syntax

C++

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

Members

`SemanticName`

Type: **LPCSTR**

The HLSL semantic associated with this element in a shader input-signature.

`SemanticIndex`

Type: **UINT**

The semantic index for the element. A semantic index modifies a semantic, with an integer index number. A semantic index is only needed in a case where there is more than one element with the same semantic. For example, a 4x4 matrix would have four components each with the semantic name **matrix**, however each of the four component would have different semantic indices (0, 1, 2, and 3).

`Format`

Type: **DXGI_FORMAT**

The data type of the element data. See DXGI_FORMAT.

`InputSlot`

Type: **UINT**

An integer value that identifies the input-assembler (see input slot). Valid values are between 0 and 15, defined in D3D10.h.

`AlignedByteOffset`

Type: **UINT**

Optional. Offset (in bytes) between each element. Use D3D10_APPEND_ALIGNED_ELEMENT for convenience to define the current element directly after the previous one, including any packing if necessary.

`InputSlotClass`

Type: **D3D10_INPUT_CLASSIFICATION**

Identifies the input data class for a single input slot (see D3D10_INPUT_CLASSIFICATION).

`InstanceDataStepRate`

Type: **UINT**

The number of instances to draw before stepping one unit forward in a vertex buffer filled with instance data. Can be any unsigned integer value (0 means do not step) when the slot class is D3D10_INPUT_PER_INSTANCE_DATA; must be 0 when the slot class is D3D10_INPUT_PER_VERTEX_DATA.

Remarks

An input-layout object contains an array of structures, each structure defines one element being read from an input slot. Create an input-layout object by calling ID3D10Device::CreateInputLayout. For an example, see Create an input-layout object.

Requirements

Header	d3d10.h

See also

Core Structures

D3D10_MESSAGE structure (d3d10sdklayers.h)

Article • 07/27/20222 minutes to read

A debug message in the Information Queue.

Syntax

C++

```
typedef struct D3D10_MESSAGE {
  D3D10_MESSAGE_CATEGORY Category;
  D3D10_MESSAGE_SEVERITY Severity;
  D3D10_MESSAGE_ID       ID;
  const char             *pDescription;
  SIZE_T                 DescriptionByteLength;
} D3D10_MESSAGE;
```

Members

`Category`

Type: **D3D10_MESSAGE_CATEGORY**

The category of the message. See D3D10_MESSAGE_CATEGORY.

`Severity`

Type: **D3D10_MESSAGE_SEVERITY**

The severity of the message. See D3D10_MESSAGE_SEVERITY.

`ID`

Type: **D3D10_MESSAGE_ID**

The ID of the message. See D3D10_MESSAGE_ID.

`pDescription`

Type: **const char***

The message string.

`DescriptionByteLength`

Type: **SIZE_T**

The length of pDescription in bytes.

Remarks

This structure is returned from ID3D10InfoQueue::GetMessage as part of the Information Queue feature (see ID3D10InfoQueue Interface).

Requirements

Header	d3d10sdklayers.h (include D3D10.h)

See also

Core Structures

D3D10_QUERY_DATA_PIPELINE_STATISTICS structure (d3d10.h)

Article • 04/02/20212 minutes to read

Query information about graphics-pipeline activity in between calls to ID3D10Asynchronous::Begin and ID3D10Asynchronous::End.

Syntax

```C++
typedef struct D3D10_QUERY_DATA_PIPELINE_STATISTICS {
  UINT64 IAVertices;
  UINT64 IAPrimitives;
  UINT64 VSInvocations;
  UINT64 GSInvocations;
  UINT64 GSPrimitives;
  UINT64 CInvocations;
  UINT64 CPrimitives;
  UINT64 PSInvocations;
} D3D10_QUERY_DATA_PIPELINE_STATISTICS;
```

Members

`IAVertices`

Type: **UINT64**

Number of vertices read by input assembler.

`IAPrimitives`

Type: **UINT64**

Number of primitives read by the input assembler. This number can be different depending on the primitive topology used. For example, a triangle strip with 6 vertices will produce 4 triangles, however a triangle list with 6 vertices will produce 2 triangles.

`VSInvocations`

Type: **UINT64**

Number of times a vertex shader was invoked. Direct3D invokes the vertex shader once per vertex.

`GSInvocations`

Type: **UINT64**

Number of times a geometry shader was invoked. When the geometry shader is set to **NULL**, this statistic may or may not increment depending on the hardware manufacturer.

`GSPrimitives`

Type: **UINT64**

Number of primitives output by a geometry shader.

`CInvocations`

Type: **UINT64**

Number of primitives that were sent to the rasterizer. When the rasterizer is disabled, this will not increment.

`CPrimitives`

Type: **UINT64**

Number of primitives that were rendered. This may be larger or smaller than CInvocations because after a primitive is clipped sometimes it is either broken up into more than one primitive or completely culled.

`PSInvocations`

Type: **UINT64**

Number of times a pixel shader was invoked.

Requirements

Header	d3d10.h

See also

D3D10_QUERY_DATA_SO_STATISTICS structure (d3d10.h)

Article • 04/02/20212 minutes to read

Query information about the amount of data streamed out to the stream-output buffers in between ID3D10Asynchronous::Begin and ID3D10Asynchronous::End.

Syntax

```
C++

typedef struct D3D10_QUERY_DATA_SO_STATISTICS {
  UINT64 NumPrimitivesWritten;
  UINT64 PrimitivesStorageNeeded;
} D3D10_QUERY_DATA_SO_STATISTICS;
```

Members

`NumPrimitivesWritten`

Type: **UINT64**

Number of primitives (that is, points, lines, and triangles) written to the stream-output buffers.

`PrimitivesStorageNeeded`

Type: **UINT64**

Number of primitives that would have been written to the stream-output buffers if there had been enough space for them all.

Requirements

Header	d3d10.h

See also

D3D10_QUERY_DATA_TIMESTAMP_DISJOINT structure (d3d10.h)

Article • 07/27/20222 minutes to read

Query information about the reliability of a timestamp query.

Syntax

C++

```
typedef struct D3D10_QUERY_DATA_TIMESTAMP_DISJOINT {
  UINT64 Frequency;
  BOOL   Disjoint;
} D3D10_QUERY_DATA_TIMESTAMP_DISJOINT;
```

Members

`Frequency`

Type: **UINT64**

How frequently the GPU counter increments in Hz.

`Disjoint`

Type: **BOOL**

If this is **TRUE**, something occurred in between the query's ID3D10Asynchronous::Begin and ID3D10Asynchronous::End calls that caused the timestamp counter to become discontinuous or disjoint, such as unplugging the AC chord on a laptop, overheating, or throttling up/down due to laptop savings events. The timestamp returned by ID3D10Asynchronous::GetData for a timestamp query is only reliable if Disjoint is **FALSE**.

Remarks

For a list of query types see D3D10_QUERY.

Requirements

Header	d3d10.h

See also

Core Structures

D3D10_QUERY_DESC structure (d3d10.h)

Article • 07/27/20222 minutes to read

Describes a query.

Syntax

```
C++

typedef struct D3D10_QUERY_DESC {
  D3D10_QUERY Query;
  UINT        MiscFlags;
} D3D10_QUERY_DESC;
```

Members

`Query`

Type: **D3D10_QUERY**

Type of query (see D3D10_QUERY).

`MiscFlags`

Type: **UINT**

Miscellaneous flags (see D3D10_QUERY_MISC_FLAG).

Requirements

Header	d3d10.h

See also

Core Structures

D3D10_RASTERIZER_DESC structure (d3d10.h)

Article10/06/2021

Describes the rasterizer state.

Syntax

C++

```
typedef struct D3D10_RASTERIZER_DESC {
  D3D10_FILL_MODE FillMode;
  D3D10_CULL_MODE CullMode;
  BOOL            FrontCounterClockwise;
  INT             DepthBias;
  FLOAT           DepthBiasClamp;
  FLOAT           SlopeScaledDepthBias;
  BOOL            DepthClipEnable;
  BOOL            ScissorEnable;
  BOOL            MultisampleEnable;
  BOOL            AntialiasedLineEnable;
} D3D10_RASTERIZER_DESC;
```

Members

`FillMode`

Type: **D3D10_FILL_MODE**

A member of the D3D10_FILL_MODE enumerated type that determines the fill mode to use when rendering. The default value is **D3D10_FILL_SOLID**.

`CullMode`

Type: **D3D10_CULL_MODE**

A member of the D3D10_CULL_MODE enumerated type that indicates whether triangles facing the specified direction are drawn. The default value is **D3D10_CULL_BACK**.

`FrontCounterClockwise`

Type: **BOOL**

Determines if a triangle is front-facing or back-facing. If this parameter is **TRUE**, then a triangle is considered front-facing if its vertices are counter-clockwise on the render target, and considered back-facing if they are clockwise. If this parameter is **FALSE**, then the opposite is true. The default value is **FALSE**.

`DepthBias`

Type: **INT**

Specifies the depth value added to a given pixel. The default value is 0. For info about depth bias, see Depth Bias.

`DepthBiasClamp`

Type: **FLOAT**

Specifies the maximum depth bias of a pixel. The default value is 0.0f. For info about depth bias, see Depth Bias.

`SlopeScaledDepthBias`

Type: **FLOAT**

Specifies a scalar on a given pixel's slope. The default value is 0.0f. For info about depth bias, see Depth Bias.

`DepthClipEnable`

Type: **BOOL**

Enables or disables clipping based on distance. The default value is **TRUE**.

The hardware always performs x and y clipping of rasterized coordinates. When **DepthClipEnable** is set to the default value, the hardware also clips the z value (that is, the hardware performs the last step of the following algorithm).

syntax

```
0 &lt; w
-w &lt;= x &lt;= w (or arbitrarily wider range if implementation uses a
guard band to reduce clipping burden)
-w &lt;= y &lt;= w (or arbitrarily wider range if implementation uses a
guard band to reduce clipping burden)
0 &lt;= z &lt;= w
```

When you set **DepthClipEnable** to FALSE, the hardware skips the z clipping (that is, the last step in the preceding algorithm). However, the hardware still performs the "0 < w" clipping. When z clipping is disabled, improper depth ordering at the pixel level might result. However, when z clipping is disabled, stencil shadow implementations are simplified. In other words, you can avoid complex special-case handling for geometry that goes beyond the back clipping plane.

`ScissorEnable`

Type: **BOOL**

Enable or disables scissor-rectangle culling. All pixels outside an active scissor rectangle are culled. The default value is **FALSE**. For more information, see Set the Scissor Rectangle.

`MultisampleEnable`

Type: **BOOL**

Specifies whether to use the quadrilateral or alpha line anti-aliasing algorithm on multisample antialiasing (MSAA) render targets. The default value is **FALSE**. Set to **TRUE** to use the quadrilateral line anti-aliasing algorithm and to **FALSE** to use the alpha line anti-aliasing algorithm. For more info about this member, see Remarks.

`AntialiasedLineEnable`

Type: **BOOL**

Specifies whether to enable line antialiasing; only applies when alpha blending is enabled, you are drawing lines, and the **MultisampleEnable** member is **FALSE**. The default value is **FALSE**. For more info about this member, see Remarks.

Remarks

Rasterizer state defines the behavior of the rasterizer stage. To create a rasterizer-state object, call ID3D10Device::CreateRasterizerState. To set rasterizer state, call ID3D10Device::RSSetState.

> **Note** For **feature levels** 9.1, 9.2, 9.3, and 10.0, if you set **MultisampleEnable** to **FALSE**, the runtime renders all points, lines, and triangles without anti-aliasing even for render targets with a sample count greater than 1. For feature level 10.1, the setting of **MultisampleEnable** has no effect on points and triangles with regard to

MSAA and impacts only the selection of the line-rendering algorithm as shown in this table:

Line-rendering algorithm	MultisampleEnable	AntialiasedLineEnable
Aliased	**FALSE**	**FALSE**
Alpha antialiased	**FALSE**	**TRUE**
Quadrilateral	**TRUE**	**FALSE**
Quadrilateral	**TRUE**	**TRUE**

The settings of the **MultisampleEnable** and **AntialiasedLineEnable** members apply only to multisample antialiasing (MSAA) render targets (that is, render targets with sample counts greater than 1). Because of the differences in feature-level behavior and as long as you aren't performing any line drawing or don't mind that lines render as quadrilaterals, we recommend that you always set **MultisampleEnable** to **TRUE** whenever you render on MSAA render targets.

Requirements

Header	d3d10.h

See also

Core Structures

D3D10_RECT

Article • 03/15/2021 • 2 minutes to read

D3D10_RECT is declared as follows:

syntax

```
typedef RECT D3D10_RECT;
```

For more information about this GDI rectangle structure, see RECT.

Remarks

This structure is used for scissor rectangles by **ID3D10Device::RSGetScissorRects** and **ID3D10Device::RSSetScissorRects**.

Requirements

Requirement	Value
Header	D3D10.h
Library	D3D10.lib

See also

Core Structures

D3D10_RENDER_TARGET_BLEND_DESC1 structure (d3d10_1.h)

Article • 04/02/20212 minutes to read

Describes the blend state for a render target for a Direct3D 10.1 device

Syntax

C++

```
typedef struct D3D10_RENDER_TARGET_BLEND_DESC1 {
  BOOL           BlendEnable;
  D3D10_BLEND    SrcBlend;
  D3D10_BLEND    DestBlend;
  D3D10_BLEND_OP BlendOp;
  D3D10_BLEND    SrcBlendAlpha;
  D3D10_BLEND    DestBlendAlpha;
  D3D10_BLEND_OP BlendOpAlpha;
  UINT8          RenderTargetWriteMask;
} D3D10_RENDER_TARGET_BLEND_DESC1;
```

Members

`BlendEnable`

Type: **BOOL**

Enable (or disable) blending.

`SrcBlend`

Type: **D3D10_BLEND**

This blend option specifies the first RGB data source and includes an optional pre-blend operation.

`DestBlend`

Type: **D3D10_BLEND**

This blend option specifies the second RGB data source and includes an optional pre-blend operation.

`BlendOp`

Type: **D3D10_BLEND_OP**

This blend operation defines how to combine the RGB data sources.

`SrcBlendAlpha`

Type: **D3D10_BLEND**

This blend option specifies the first alpha data source and includes an optional pre-blend operation. Blend options that end in _COLOR are not allowed.

`DestBlendAlpha`

Type: **D3D10_BLEND**

This blend option specifies the second alpha data source and includes an optional pre-blend operation. Blend options that end in _COLOR are not allowed.

`BlendOpAlpha`

Type: **D3D10_BLEND_OP**

This blend operation defines how to combine the alpha data sources.

`RenderTargetWriteMask`

Type: **UINT8**

A write mask.

Remarks

To see how blending is done, see Output-Merger Stage (Direct3D 10).

These are the default values for blend state.

State	Default Value
BlendEnable	FALSE
SrcBlend	D3D10_BLEND_ONE
DestBlend	D3D10_BLEND_ZERO
BlendOp	D3D10_BLEND_OP_ADD

SrcBlendAlpha	D3D10_BLEND_ONE
DestBlendAlpha	D3D10_BLEND_ZERO
BlendOpAlpha	D3D10_BLEND_OP_ADD
RenderTargetWriteMask	D3D10_COLOR_WRITE_ENABLE_ALL

Requirements

Header	d3d10_1.h (include D3D10.h)

See also

Core Structures

D3D10_SAMPLER_DESC structure (d3d10.h)

Article09/01/2022

Describes a sampler state.

Syntax

C++

```
typedef struct D3D10_SAMPLER_DESC {
  D3D10_FILTER               Filter;
  D3D10_TEXTURE_ADDRESS_MODE AddressU;
  D3D10_TEXTURE_ADDRESS_MODE AddressV;
  D3D10_TEXTURE_ADDRESS_MODE AddressW;
  FLOAT                      MipLODBias;
  UINT                       MaxAnisotropy;
  D3D10_COMPARISON_FUNC      ComparisonFunc;
  FLOAT                      BorderColor[4];
  FLOAT                      MinLOD;
  FLOAT                      MaxLOD;
} D3D10_SAMPLER_DESC;
```

Members

`Filter`

Type: **D3D10_FILTER**

Filtering method to use when sampling a texture (see D3D10_FILTER).

`AddressU`

Type: **D3D10_TEXTURE_ADDRESS_MODE**

Method to use for resolving a u texture coordinate that is outside the 0 to 1 range (see D3D10_TEXTURE_ADDRESS_MODE).

`AddressV`

Type: **D3D10_TEXTURE_ADDRESS_MODE**

Method to use for resolving a v texture coordinate that is outside the 0 to 1 range.

`AddressW`

Type: **D3D10_TEXTURE_ADDRESS_MODE**

Method to use for resolving a w texture coordinate that is outside the 0 to 1 range.

`MipLODBias`

Type: **FLOAT**

Offset from the calculated mipmap level. For example, if Direct3D calculates that a texture should be sampled at mipmap level 3 and MipLODBias is 2, then the texture will be sampled at mipmap level 5.

`MaxAnisotropy`

Type: **UINT**

Clamping value used if D3D10_FILTER_ANISOTROPIC or D3D10_FILTER_COMPARISON_ANISOTROPIC is specified in Filter. Valid values are between 1 and 16.

`ComparisonFunc`

Type: **D3D10_COMPARISON_FUNC**

A function that compares sampled data against existing sampled data. The function options are listed in D3D10_COMPARISON_FUNC.

`BorderColor[4]`

Type: **FLOAT**

Border color to use if D3D10_TEXTURE_ADDRESS_BORDER is specified for AddressU, AddressV, or AddressW. Range must be between 0.0 and 1.0 inclusive.

`MinLOD`

Type: **FLOAT**

Lower end of the mipmap range to clamp access to, where 0 is the largest and most detailed mipmap level and any level higher than that is less detailed.

`MaxLOD`

Type: **FLOAT**

Upper end of the mipmap range to clamp access to, where 0 is the largest and most detailed mipmap level and any level higher than that is less detailed. This value must be greater than or equal to MinLOD. To have no upper limit on LOD set this to a large value such as D3D10_FLOAT32_MAX.

Remarks

These are the default values for sampler state.

State	Default Value
Filter	Min_Mag_Mip_Point
AddressU	Clamp
AddressV	Clamp
AddressW	Clamp
MinLOD	0.0f
MaxLOD	3.402823466e+38F (FLT_MAX)
MipMapLODBias	0.0f
MaxAnisotropy	16
ComparisonFunc	Never
BorderColor	float4(0.0f, 0.0f, 0.0f, 0.0f)
Texture	N/A

Requirements

Header	d3d10.h

See also

Core Structures

D3D10_SHADER_RESOURCE_VIEW_DESC structure (d3d10.h)

Article • 07/27/20222 minutes to read

Describes a shader-resource view.

Syntax

C++

```
typedef struct D3D10_SHADER_RESOURCE_VIEW_DESC {
  DXGI_FORMAT         Format;
  D3D10_SRV_DIMENSION ViewDimension;
  union {
    D3D10_BUFFER_SRV        Buffer;
    D3D10_TEX1D_SRV         Texture1D;
    D3D10_TEX1D_ARRAY_SRV   Texture1DArray;
    D3D10_TEX2D_SRV         Texture2D;
    D3D10_TEX2D_ARRAY_SRV   Texture2DArray;
    D3D10_TEX2DMS_SRV       Texture2DMS;
    D3D10_TEX2DMS_ARRAY_SRV Texture2DMSArray;
    D3D10_TEX3D_SRV         Texture3D;
    D3D10_TEXCUBE_SRV       TextureCube;
  };
} D3D10_SHADER_RESOURCE_VIEW_DESC;
```

Members

`Format`

Type: **DXGI_FORMAT**

The viewing format. See remarks.

`ViewDimension`

Type: **D3D10_SRV_DIMENSION**

The resource type of the view. See D3D10_SRV_DIMENSION. This should be the same as the resource type of the underlying resource. This parameter also determines which _SRV to use in the union below.

`Buffer`

Type: **D3D10_BUFFER_SRV**

View the resource as a buffer using information from a shader-resource view (see D3D10_BUFFER_SRV).

`Texture1D`

Type: **D3D10_TEX1D_SRV**

View the resource as a 1D texture using information from a shader-resource view (see D3D10_TEX1D_SRV).

`Texture1DArray`

Type: **D3D10_TEX1D_ARRAY_SRV**

View the resource as a 1D-texture array using information from a shader-resource view (see D3D10_TEX1D_ARRAY_SRV.

`Texture2D`

Type: **D3D10_TEX2D_SRV**

View the resource as a 2D-texture using information from a shader-resource view (see D3D10_TEX2D_SRV.

`Texture2DArray`

Type: **D3D10_TEX2D_ARRAY_SRV**

View the resource as a 2D-texture array using information from a shader-resource view (see D3D10_TEX2D_ARRAY_SRV.

`Texture2DMS`

Type: **D3D10_TEX2DMS_SRV**

View the resource as a 2D-multisampled texture using information from a shader-resource view (see D3D10_TEX2DMS_SRV.

`Texture2DMSArray`

Type: **D3D10_TEX2DMS_ARRAY_SRV**

View the resource as a 2D-multisampled-texture array using information from a shader-resource view (see D3D10_TEX2DMS_ARRAY_SRV.

`Texture3D`

Type: **D3D10_TEX3D_SRV**

View the resource as a 3D texture using information from a shader-resource view (see D3D10_TEX3D_SRV.

`TextureCube`

Type: **D3D10_TEXCUBE_SRV**

View the resource as a 3D-cube texture using information from a shader-resource view (see D3D10_TEXCUBE_SRV).

Remarks

A view is a format-specific way to look at the data in a resource. The view determines what data to look at, and how it is cast when read. For more information about how views work, see Views

When viewing a resource, the resource-view description must specify a typed format, that is compatible with the resource format. So that means that you cannot create a resource-view description using any format with _TYPELESS in the name. You can however view a typeless resource by specifying a typed format for the view. For example, a DXGI_FORMAT_R32G32B32_TYPELESS resource can be viewed with one of these typed formats: DXGI_FORMAT_R32G32B32_FLOAT, DXGI_FORMAT_R32G32B32_UINT, and DXGI_FORMAT_R32G32B32_SINT, since these typed formats are compatible with the typeless resource.

Create a shader-resource-view description by calling ID3D10Device::CreateShaderResourceView. To view a shader-resource-view description, call ID3D10ShaderResourceView::GetDesc.

Requirements

Header	d3d10.h (include D3D10Shader.h)

See also

Core Structures

D3D10_SHADER_RESOURCE_VIEW_DESC1 structure (d3d10_1.h)

Article • 07/27/20222 minutes to read

Describes a shader-resource view.

Syntax

C++

```
typedef struct D3D10_SHADER_RESOURCE_VIEW_DESC1 {
  DXGI_FORMAT           Format;
  D3D10_SRV_DIMENSION1 ViewDimension;
  union {
    D3D10_BUFFER_SRV         Buffer;
    D3D10_TEX1D_SRV          Texture1D;
    D3D10_TEX1D_ARRAY_SRV    Texture1DArray;
    D3D10_TEX2D_SRV          Texture2D;
    D3D10_TEX2D_ARRAY_SRV    Texture2DArray;
    D3D10_TEX2DMS_SRV        Texture2DMS;
    D3D10_TEX2DMS_ARRAY_SRV  Texture2DMSArray;
    D3D10_TEX3D_SRV          Texture3D;
    D3D10_TEXCUBE_SRV        TextureCube;
    D3D10_TEXCUBE_ARRAY_SRV1 TextureCubeArray;
  };
} D3D10_SHADER_RESOURCE_VIEW_DESC1;
```

Members

`Format`

Type: **DXGI_FORMAT**

The viewing format. See remarks.

`ViewDimension`

Type: **D3D10_SRV_DIMENSION1**

The resource type of the view. See D3D10_SRV_DIMENSION1. This should be the same as the resource type of the underlying resource. This parameter also determines which _SRV to use in the union below.

`Buffer`

Type: **D3D10_BUFFER_SRV**

View the resource as a buffer using information from a shader-resource view (see D3D10_BUFFER_SRV).

`Texture1D`

Type: **D3D10_TEX1D_SRV**

View the resource as a 1D texture using information from a shader-resource view (see D3D10_TEX1D_SRV).

`Texture1DArray`

Type: **D3D10_TEX1D_ARRAY_SRV**

View the resource as a 1D-texture array using information from a shader-resource view (see D3D10_TEX1D_ARRAY_SRV.

`Texture2D`

Type: **D3D10_TEX2D_SRV**

View the resource as a 2D-texture using information from a shader-resource view (see D3D10_TEX2D_SRV.

`Texture2DArray`

Type: **D3D10_TEX2D_ARRAY_SRV**

View the resource as a 2D-texture array using information from a shader-resource view (see D3D10_TEX2D_ARRAY_SRV.

`Texture2DMS`

Type: **D3D10_TEX2DMS_SRV**

View the resource as a 2D-multisampled texture using information from a shader-resource view (see D3D10_TEX2DMS_SRV.

`Texture2DMSArray`

Type: **D3D10_TEX2DMS_ARRAY_SRV**

View the resource as a 2D-multisampled-texture array using information from a shader-resource view (see D3D10_TEX2DMS_ARRAY_SRV.

`Texture3D`

Type: **D3D10_TEX3D_SRV**

View the resource as a 3D texture using information from a shader-resource view (see D3D10_TEX3D_SRV.

`TextureCube`

Type: **D3D10_TEXCUBE_SRV**

View the resource as a 3D-cube texture using information from a shader-resource view (see D3D10_TEXCUBE_SRV).

Type: **D3D10_TEXCUBE_ARRAY_SRV1**

View the resource as an array of cube textures using information from a shader-resource view (see D3D10_TEXCUBE_ARRAY_SRV1).

`TextureCubeArray`

Remarks

A view is a format-specific way to look at the data in a resource. The view determines what data to look at, and how it is cast when read. For more information about how views work, see Views

When viewing a resource, the resource-view description must specify a typed format, that is compatible with the resource format. So that means that you cannot create a resource-view description using any format with _TYPELESS in the name. You can however view a typeless resource by specifying a typed format for the view. For example, a DXGI_FORMAT_R32G32B32_TYPELESS resource can be viewed with one of these typed formats: DXGI_FORMAT_R32G32B32_FLOAT, DXGI_FORMAT_R32G32B32_UINT, and DXGI_FORMAT_R32G32B32_SINT, since these typed formats are compatible with the typeless resource.

Create a shader-resource-view description by calling ID3D10Device1::CreateShaderResourceView1. To view a shader-resource-view description, call ID3D10ShaderResourceView::GetDesc.

This structure requires Windows Vista Service Pack 1.

Requirements

Header	d3d10_1.h (include D3D10_1Shader.h)

See also

Core Structures

Shader Structures

D3D10_STATE_BLOCK_MASK structure (d3d10effect.h)

Article • 09/01/20222 minutes to read

Indicates the device state.

Syntax

C++

```
typedef struct _D3D10_STATE_BLOCK_MASK {
  BYTE VS;
  BYTE
VSSamplers[D3D10_BYTES_FROM_BITS(D3D10_COMMONSHADER_SAMPLER_SLOT_COUNT)];
  BYTE
VSShaderResources[D3D10_BYTES_FROM_BITS(D3D10_COMMONSHADER_INPUT_RESOURCE_SL
OT_COUNT)];
  BYTE
VSConstantBuffers[D3D10_BYTES_FROM_BITS(D3D10_COMMONSHADER_CONSTANT_BUFFER_A
PI_SLOT_COUNT)];
  BYTE GS;
  BYTE
GSSamplers[D3D10_BYTES_FROM_BITS(D3D10_COMMONSHADER_SAMPLER_SLOT_COUNT)];
  BYTE
GSShaderResources[D3D10_BYTES_FROM_BITS(D3D10_COMMONSHADER_INPUT_RESOURCE_SL
OT_COUNT)];
  BYTE
GSConstantBuffers[D3D10_BYTES_FROM_BITS(D3D10_COMMONSHADER_CONSTANT_BUFFER_A
PI_SLOT_COUNT)];
  BYTE PS;
  BYTE
PSSamplers[D3D10_BYTES_FROM_BITS(D3D10_COMMONSHADER_SAMPLER_SLOT_COUNT)];
  BYTE
PSShaderResources[D3D10_BYTES_FROM_BITS(D3D10_COMMONSHADER_INPUT_RESOURCE_SL
OT_COUNT)];
  BYTE
PSConstantBuffers[D3D10_BYTES_FROM_BITS(D3D10_COMMONSHADER_CONSTANT_BUFFER_A
PI_SLOT_COUNT)];
  BYTE
IAVertexBuffers[D3D10_BYTES_FROM_BITS(D3D10_IA_VERTEX_INPUT_RESOURCE_SLOT_CO
UNT)];
  BYTE IAIndexBuffer;
  BYTE IAInputLayout;
  BYTE IAPrimitiveTopology;
  BYTE OMRenderTargets;
  BYTE OMDepthStencilState;
  BYTE OMBlendState;
  BYTE RSViewports;
  BYTE RSScissorRects;
```

```
    BYTE RSRasterizerState;
    BYTE SOBuffers;
    BYTE Predication;
  } D3D10_STATE_BLOCK_MASK;
```

Members

`VS`

Type: **BYTE**

Boolean value indicating whether to save the vertex shader state.

`VSSamplers[D3D10_BYTES_FROM_BITS(D3D10_COMMONSHADER_SAMPLER_SLOT_COUNT)]`

Type: **BYTE**

Array of vertex-shader samplers. The array is a multi-byte bitmask where each bit represents one sampler slot.

`VSShaderResources[D3D10_BYTES_FROM_BITS(D3D10_COMMONSHADER_INPUT_RESOURCE_SLOT_COUNT)]`

Type: **BYTE**

Array of vertex-shader resources. The array is a multi-byte bitmask where each bit represents one resource slot.

`VSConstantBuffers[D3D10_BYTES_FROM_BITS(D3D10_COMMONSHADER_CONSTANT_BUFFER_API_SLOT_COUNT)]`

Type: **BYTE**

Array of vertex-shader constant buffers. The array is a multi-byte bitmask where each bit represents one constant buffer slot.

`GS`

Type: **BYTE**

Boolean value indicating whether to save the geometry shader state.

`GSSamplers[D3D10_BYTES_FROM_BITS(D3D10_COMMONSHADER_SAMPLER_SLOT_COUNT)]`

Type: **BYTE**

Array of geometry-shader samplers. The array is a multi-byte bitmask where each bit represents one sampler slot.

`GSShaderResources[D3D10_BYTES_FROM_BITS(D3D10_COMMONSHADER_INPUT_RESOURCE_SLOT_COUNT)]`

Type: **BYTE**

Array of geometry-shader resources. The array is a multi-byte bitmask where each bit represents one resource slot.

`GSConstantBuffers[D3D10_BYTES_FROM_BITS(D3D10_COMMONSHADER_CONSTANT_BUFFER_API_SLOT_COUNT)]`

Type: **BYTE**

Array of geometry-shader constant buffers. The array is a multi-byte bitmask where each bit represents one buffer slot.

`PS`

Type: **BYTE**

Boolean value indicating whether to save the pixel shader state.

`PSSamplers[D3D10_BYTES_FROM_BITS(D3D10_COMMONSHADER_SAMPLER_SLOT_COUNT)]`

Type: **BYTE**

Array of pixel-shader samplers. The array is a multi-byte bitmask where each bit represents one sampler slot.

`PSShaderResources[D3D10_BYTES_FROM_BITS(D3D10_COMMONSHADER_INPUT_RESOURCE_SLOT_COUNT)]`

Type: **BYTE**

Array of pixel-shader resources. The array is a multi-byte bitmask where each bit represents one resource slot.

`PSConstantBuffers[D3D10_BYTES_FROM_BITS(D3D10_COMMONSHADER_CONSTANT_BUFFER_API_SLOT_COUNT)]`

Type: **BYTE**

Array of pixel-shader constant buffers. The array is a multi-byte bitmask where each bit represents one constant buffer slot.

`IAVertexBuffers[D3D10_BYTES_FROM_BITS(D3D10_IA_VERTEX_INPUT_RESOURCE_SLOT_COUNT)]`

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. The D3D10StateBlockMaskEnableCapture function provides a convenient way of setting a range of bitmasks for the array members of **D3D10_STATE_BLOCK_MASK**.

Requirements

Header	d3d10effect.h (include D3D10.h)

See also

Core Structures

D3D10_SO_DECLARATION_ENTRY structure (d3d10.h)

Article • 07/27/20222 minutes to read

Description of a vertex element in a vertex buffer in an output slot.

Syntax

C++

```
typedef struct D3D10_SO_DECLARATION_ENTRY {
  LPCSTR SemanticName;
  UINT   SemanticIndex;
  BYTE   StartComponent;
  BYTE   ComponentCount;
  BYTE   OutputSlot;
} D3D10_SO_DECLARATION_ENTRY;
```

Members

`SemanticName`

Type: **LPCSTR**

Type of output element. Possible values: "POSITION", "NORMAL", or "TEXCOORD0".

`SemanticIndex`

Type: **UINT**

Output element's zero-based index. Should be used if, for example, you have more than one texture coordinate stored in each vertex.

`StartComponent`

Type: **BYTE**

Which component of the entry to begin writing out to. Valid values are 0 ~ 3. For example, if you only wish to output to the y and z components of a position, then StartComponent should be 1 and ComponentCount should be 2.

`ComponentCount`

Type: **BYTE**

The number of components of the entry to write out to. Valid values are 1 ~ 4. For example, if you only wish to output to the y and z components of a position, then StartComponent should be 1 and ComponentCount should be 2.

`OutputSlot`

Type: **BYTE**

The output slot that contains the vertex buffer that contains this output entry.

Requirements

Header	d3d10.h

See also

Core Structures

D3D10_VIEWPORT structure (d3d10.h)

Article • 07/27/20222 minutes to read

Defines the dimensions of a viewport.

Syntax

C++

```
typedef struct D3D10_VIEWPORT {
  INT   TopLeftX;
  INT   TopLeftY;
  UINT  Width;
  UINT  Height;
  FLOAT MinDepth;
  FLOAT MaxDepth;
} D3D10_VIEWPORT;
```

Members

`TopLeftX`

Type: **INT**

X position of the left hand side of the viewport. Ranges between D3D10_VIEWPORT_BOUNDS_MIN and D3D10_VIEWPORT_BOUNDS_MAX.

`TopLeftY`

Type: **INT**

Y position of the top of the viewport. Ranges between D3D10_VIEWPORT_BOUNDS_MIN and D3D10_VIEWPORT_BOUNDS_MAX.

`Width`

Type: **UINT**

Width of the viewport.

`Height`

Type: **UINT**

Height of the viewport.

`MinDepth`

Type: **FLOAT**

Minimum depth of the viewport. Ranges between 0 and 1.

`MaxDepth`

Type: **FLOAT**

Maximum depth of the viewport. Ranges between 0 and 1.

Remarks

In all cases, **Width** and **Height** must be ≥ 0 and **TopLeftX** + **Width** and **TopLeftY** + **Height** must be ≤ D3D10_VIEWPORT_BOUNDS_MAX.

Requirements

Header	d3d10.h

See also

Core Structures

Direct3D 10 core enumerations

Article • 01/06/2021 • 2 minutes to read

This section contains information about the following core enumerations:

Enumeration	Description
D3D10_ASYNC_GETDATA_FLAG	Optional flags that control what happens when data is retrieved by calling ID3D10Asynchronous::GetData.
D3D10_BLEND	Blend options.
D3D10_BLEND_OP	Blending operations between source and destination pixels.
D3D10_CLEAR_FLAG	Specifies which parts of the depth stencil to clear.
D3D10_COLOR_WRITE_ENABLE	Color writing masks.
D3D10_COMPARISON_FUNC	Comparison functions.
D3D10_COUNTER	Types of performance counters.
D3D10_COUNTER_TYPE	Data type of a counter.
D3D10_CREATE_DEVICE_FLAG	Device creation flags.
D3D10_CULL_MODE	Rasterizer cull modes.
D3D10_DEPTH_WRITE_MASK	Determines which portion of the depth stencil is writable.
D3D10_DEVICE_STATE_TYPES	Used in ID3D10StateBlock Interface function calls.
D3D10_DRIVER_TYPE	Type of device driver.
D3D10_FEATURE_LEVEL1	The version of hardware acceleration.
D3D10_FILL_MODE	Rasterizer fill modes.
D3D10_FILTER	Types of sampler filters.
D3D10_FILTER_TYPE	Types of texture-sampling filters.
D3D10_FORMAT_SUPPORT	Which resources are supported for a given format and given device. See ID3D10Device::CheckFormatSupport.
D3D10_INPUT_CLASSIFICATION	Types of input data.
D3D10_MESSAGE_CATEGORY	Categories of debug messages.

Enumeration	Description
D3D10_MESSAGE_ID	A debug message's unique ID.
D3D10_MESSAGE_SEVERITY	Severity of a debug message.
D3D10_PRIMITIVE_TOPOLOGY	Types of primitive topology or the way a primitive's data is arranged.
D3D10_QUERY	Types of queries.
D3D10_QUERY_MISC_FLAG	Flags that describe miscellaneous query behavior.
D3D10_RAISE_FLAG	Flags that indicate how to handle internal driver errors.
D3D10_REGISTER_COMPONENT_TYPE	The register component types, usually used in D3D10_SIGNATURE_PARAMETER_DESC.
D3D10_RESOURCE_RETURN_TYPE	The return type of a resource. See D3D10_SHADER_INPUT_BIND_DESC.
D3D10_STENCIL_OP	Stencil operations.
D3D10_TEXTURE_ADDRESS_MODE	Actions to perform when a texture coordinate is outside of the boundaries of a texture.
D3D10_TEXTURECUBE_FACE	The different faces of a cube texture.

Related topics

Core Reference

D3D10_ASYNC_GETDATA_FLAG enumeration (d3d10.h)

Article • 01/31/20222 minutes to read

Optional flags that control the behavior of ID3D10Asynchronous::GetData.

Syntax

C++

```
typedef enum D3D10_ASYNC_GETDATA_FLAG {
  D3D10_ASYNC_GETDATA_DONOTFLUSH = 0x1
} ;
```

Constants

`D3D10_ASYNC_GETDATA_DONOTFLUSH` Value: *0x1* Do not flush the command buffer. This can potentially cause an infinite loop if GetData is continually called until it returns S_OK as there may still be commands in the command buffer that need to be processed in order for GetData to return S_OK. Since the commands in the command buffer are not flushed they will not be processed and therefore GetData will never return S_OK.

Requirements

Header	d3d10.h

See also

Core Enumerations

D3D10_BLEND enumeration (d3d10.h)

Article • 01/31/20222 minutes to read

Blend options. A blend option identifies the data source and an optional pre-blend operation.

Syntax

C++

```
typedef enum D3D10_BLEND {
  D3D10_BLEND_ZERO = 1,
  D3D10_BLEND_ONE = 2,
  D3D10_BLEND_SRC_COLOR = 3,
  D3D10_BLEND_INV_SRC_COLOR = 4,
  D3D10_BLEND_SRC_ALPHA = 5,
  D3D10_BLEND_INV_SRC_ALPHA = 6,
  D3D10_BLEND_DEST_ALPHA = 7,
  D3D10_BLEND_INV_DEST_ALPHA = 8,
  D3D10_BLEND_DEST_COLOR = 9,
  D3D10_BLEND_INV_DEST_COLOR = 10,
  D3D10_BLEND_SRC_ALPHA_SAT = 11,
  D3D10_BLEND_BLEND_FACTOR = 14,
  D3D10_BLEND_INV_BLEND_FACTOR = 15,
  D3D10_BLEND_SRC1_COLOR = 16,
  D3D10_BLEND_INV_SRC1_COLOR = 17,
  D3D10_BLEND_SRC1_ALPHA = 18,
  D3D10_BLEND_INV_SRC1_ALPHA = 19
} ;
```

Constants

`D3D10_BLEND_ZERO` Value: *1* The data source is the color black (0, 0, 0, 0). No pre-blend operation.
`D3D10_BLEND_ONE` Value: *2* The data source is the color white (1, 1, 1, 1). No pre-blend operation.
`D3D10_BLEND_SRC_COLOR` Value: *3* The data source is color data (RGB) from a pixel shader. No pre-blend operation.

`D3D10_BLEND_INV_SRC_COLOR` Value: *4* The data source is color data (RGB) from a pixel shader. The pre-blend operation inverts the data, generating 1 - RGB.
`D3D10_BLEND_SRC_ALPHA` Value: *5* The data source is alpha data (A) from a pixel shader. No pre-blend operation.
`D3D10_BLEND_INV_SRC_ALPHA` Value: *6* The data source is alpha data (A) from a pixel shader. The pre-blend operation inverts the data, generating 1 - A.
`D3D10_BLEND_DEST_ALPHA` Value: *7* The data source is alpha data from a rendertarget. No pre-blend operation.
`D3D10_BLEND_INV_DEST_ALPHA` Value: *8* The data source is alpha data from a rendertarget. The pre-blend operation inverts the data, generating 1 - A.
`D3D10_BLEND_DEST_COLOR` Value: *9* The data source is color data from a rendertarget. No pre-blend operation.
`D3D10_BLEND_INV_DEST_COLOR` Value: *10* The data source is color data from a rendertarget. The pre-blend operation inverts the data, generating 1 - RGB.
`D3D10_BLEND_SRC_ALPHA_SAT` Value: *11* The data source is alpha data from a pixel shader. The pre-blend operation clamps the data to 1 or less.
`D3D10_BLEND_BLEND_FACTOR` Value: *14* The data source is the blend factor set with ID3D10Device::OMSetBlendState. No pre-blend operation.
`D3D10_BLEND_INV_BLEND_FACTOR` Value: *15* The data source is the blend factor set with ID3D10Device::OMSetBlendState. The pre-blend operation inverts the blend factor, generating 1 - blend_factor.

`D3D10_BLEND_SRC1_COLOR` Value: *16* The data sources are both color data output by a pixel shader. There is no pre-blend operation. This options supports dual-source color blending.	
`D3D10_BLEND_INV_SRC1_COLOR` Value: *17* The data sources are both color data output by a pixel shader. The pre-blend operation inverts the data, generating 1 - RGB. This options supports dual-source color blending.	
`D3D10_BLEND_SRC1_ALPHA` Value: *18* The data sources are alpha data output by a pixel shader. There is no pre-blend operation. This options supports dual-source color blending.	
`D3D10_BLEND_INV_SRC1_ALPHA` Value: *19* The data sources are alpha data output by a pixel shader. The pre-blend operation inverts the data, generating 1 - A. This options supports dual-source color blending.	

Remarks

Blend operations are specified in a blend description.

Requirements

Header	d3d10.h

See also

Core Enumerations

D3D10_BLEND_OP enumeration (d3d10.h)

Article • 07/27/20222 minutes to read

RGB or alpha blending operation.

Syntax

C++

```
typedef enum D3D10_BLEND_OP {
  D3D10_BLEND_OP_ADD = 1,
  D3D10_BLEND_OP_SUBTRACT = 2,
  D3D10_BLEND_OP_REV_SUBTRACT = 3,
  D3D10_BLEND_OP_MIN = 4,
  D3D10_BLEND_OP_MAX = 5
} ;
```

Constants

`D3D10_BLEND_OP_ADD` Value: *1* Add source 1 and source 2.
`D3D10_BLEND_OP_SUBTRACT` Value: *2* Subtract source 1 from source 2.
`D3D10_BLEND_OP_REV_SUBTRACT` Value: *3* Subtract source 2 from source 1.
`D3D10_BLEND_OP_MIN` Value: *4* Find the minimum of source 1 and source 2.
`D3D10_BLEND_OP_MAX` Value: *5* Find the maximum of source 1 and source 2.

Remarks

The runtime implements RGB blending and alpha blending separately. Therefore, blend state requires separate blend operations for RGB data and alpha data. These blend operations are specified in a blend description. The two sources — source 1 and source 2 — are shown in the blending block diagram.

Blend state is used by the output-merger stage to determine how to blend together two RGB pixel values and two alpha values. The two RGB pixel values and two alpha values are the RGB pixel value and alpha value that the pixel shader outputs and the RGB pixel value and alpha value already in the output render target. The blend option controls the data source that the blending stage uses to modulate values for the pixel shader, render target, or both. The **blend operation** controls how the blending stage mathematically combines these modulated values.

Requirements

Header	d3d10.h

See also

Core Enumerations

D3D10_CLEAR_FLAG enumeration (d3d10.h)

Article01/31/2022

Specifies the parts of the depth stencil to clear. Usually used with ID3D10Device::ClearDepthStencilView.

Syntax

```
C++

typedef enum D3D10_CLEAR_FLAG {
  D3D10_CLEAR_DEPTH = 0x1L,
  D3D10_CLEAR_STENCIL = 0x2L
} ;
```

Constants

`D3D10_CLEAR_DEPTH` Value: *0x1L* Clear the depth buffer.
`D3D10_CLEAR_STENCIL` Value: *0x2L* Clear the stencil buffer.

Remarks

These flags can be bitwise ORed together.

Requirements

Header	d3d10.h

See also

Core Enumerations

D3D10_COLOR_WRITE_ENABLE enumeration (d3d10.h)

Article • 07/27/20222 minutes to read

Identify which components of each pixel of a render target are writable during blending.

Syntax

C++

```
typedef enum D3D10_COLOR_WRITE_ENABLE {
  D3D10_COLOR_WRITE_ENABLE_RED = 1,
  D3D10_COLOR_WRITE_ENABLE_GREEN = 2,
  D3D10_COLOR_WRITE_ENABLE_BLUE = 4,
  D3D10_COLOR_WRITE_ENABLE_ALPHA = 8,
  D3D10_COLOR_WRITE_ENABLE_ALL
} ;
```

Constants

`D3D10_COLOR_WRITE_ENABLE_RED` Value: *1* Allow data to be stored in the red component.
`D3D10_COLOR_WRITE_ENABLE_GREEN` Value: *2* Allow data to be stored in the green component.
`D3D10_COLOR_WRITE_ENABLE_BLUE` Value: *4* Allow data to be stored in the blue component.
`D3D10_COLOR_WRITE_ENABLE_ALPHA` Value: *8* Allow data to be stored in the alpha component.
`D3D10_COLOR_WRITE_ENABLE_ALL` Allow data to be stored in all components.

Remarks

These flags can be combined with a bitwise OR.

Requirements

Header	d3d10.h

See also

Core Enumerations

D3D10_COMPARISON_FUNC enumeration (d3d10.h)

Article • 07/27/20222 minutes to read

Comparison options.

Syntax

C++

```
typedef enum D3D10_COMPARISON_FUNC {
  D3D10_COMPARISON_NEVER = 1,
  D3D10_COMPARISON_LESS = 2,
  D3D10_COMPARISON_EQUAL = 3,
  D3D10_COMPARISON_LESS_EQUAL = 4,
  D3D10_COMPARISON_GREATER = 5,
  D3D10_COMPARISON_NOT_EQUAL = 6,
  D3D10_COMPARISON_GREATER_EQUAL = 7,
  D3D10_COMPARISON_ALWAYS = 8
} ;
```

Constants

`D3D10_COMPARISON_NEVER` Value: *1* Never pass the comparison.
`D3D10_COMPARISON_LESS` Value: *2* If the source data is less than the destination data, the comparison passes.
`D3D10_COMPARISON_EQUAL` Value: *3* If the source data is equal to the destination data, the comparison passes.
`D3D10_COMPARISON_LESS_EQUAL` Value: *4* If the source data is less than or equal to the destination data, the comparison passes.
`D3D10_COMPARISON_GREATER` Value: *5* If the source data is greater than the destination data, the comparison passes.

`D3D10_COMPARISON_NOT_EQUAL` Value: *6* If the source data is not equal to the destination data, the comparison passes.	
`D3D10_COMPARISON_GREATER_EQUAL` Value: *7* If the source data is greater than or equal to the destination data, the comparison passes.	
`D3D10_COMPARISON_ALWAYS` Value: *8* Always pass the comparison.	

Remarks

A comparison option determines whether how the runtime compares source (new) data against destination (existing) data before storing the new data. The comparison option is declared in a description before an object is created. The API allows you to set a comparison option for a depth-stencil buffer (see D3D10_DEPTH_STENCIL_DESC), depth-stencil operations (see D3D10_DEPTH_STENCILOP_DESC), or sampler state (see D3D10_SAMPLER_DESC).

Requirements

Header	d3d10.h

See also

Core Enumerations

D3D10_COUNTER enumeration (d3d10.h)

Article • 01/31/20222 minutes to read

Performance counter types.

Syntax

C++

```
typedef enum D3D10_COUNTER {
  D3D10_COUNTER_GPU_IDLE = 0,
  D3D10_COUNTER_VERTEX_PROCESSING,
  D3D10_COUNTER_GEOMETRY_PROCESSING,
  D3D10_COUNTER_PIXEL_PROCESSING,
  D3D10_COUNTER_OTHER_GPU_PROCESSING,
  D3D10_COUNTER_HOST_ADAPTER_BANDWIDTH_UTILIZATION,
  D3D10_COUNTER_LOCAL_VIDMEM_BANDWIDTH_UTILIZATION,
  D3D10_COUNTER_VERTEX_THROUGHPUT_UTILIZATION,
  D3D10_COUNTER_TRIANGLE_SETUP_THROUGHPUT_UTILIZATION,
  D3D10_COUNTER_FILLRATE_THROUGHPUT_UTILIZATION,
  D3D10_COUNTER_VS_MEMORY_LIMITED,
  D3D10_COUNTER_VS_COMPUTATION_LIMITED,
  D3D10_COUNTER_GS_MEMORY_LIMITED,
  D3D10_COUNTER_GS_COMPUTATION_LIMITED,
  D3D10_COUNTER_PS_MEMORY_LIMITED,
  D3D10_COUNTER_PS_COMPUTATION_LIMITED,
  D3D10_COUNTER_POST_TRANSFORM_CACHE_HIT_RATE,
  D3D10_COUNTER_TEXTURE_CACHE_HIT_RATE,
  D3D10_COUNTER_DEVICE_DEPENDENT_0 = 0x40000000
} ;
```

Constants

`D3D10_COUNTER_GPU_IDLE` Value: *0* Percentage of the time that the GPU is idle.
`D3D10_COUNTER_VERTEX_PROCESSING` Percentage of the time that the GPU does vertex processing.
`D3D10_COUNTER_GEOMETRY_PROCESSING` Percentage of the time that the GPU does geometry processing.

`D3D10_COUNTER_PIXEL_PROCESSING` Percentage of the time that the GPU does pixel processing.
`D3D10_COUNTER_OTHER_GPU_PROCESSING` Percentage of the time that the GPU does other processing (not vertex, geometry, or pixel processing).
`D3D10_COUNTER_HOST_ADAPTER_BANDWIDTH_UTILIZATION` Percentage of bandwidth used on a host adapter. Value returned by ID3D10Asynchronous::GetData between 0.0 and 1.0 when using this counter.
`D3D10_COUNTER_LOCAL_VIDMEM_BANDWIDTH_UTILIZATION` Percentage of bandwidth used by the local video memory. Value returned by ID3D10Asynchronous::GetData between 0.0 and 1.0 when using this counter
`D3D10_COUNTER_VERTEX_THROUGHPUT_UTILIZATION` Percentage of throughput used for vertices. Value returned by ID3D10Asynchronous::GetData between 0.0 and 1.0 when using this counter
`D3D10_COUNTER_TRIANGLE_SETUP_THROUGHPUT_UTILIZATION` Percentage of throughput used for triangle setup. Value returned by ID3D10Asynchronous::GetData between 0.0 and 1.0 when using this counter
`D3D10_COUNTER_FILLRATE_THROUGHPUT_UTILIZATION` Percentage of throughput used for the fillrate. Value returned by ID3D10Asynchronous::GetData between 0.0 and 1.0 when using this counter.
`D3D10_COUNTER_VS_MEMORY_LIMITED` Percentage of time that a vertex shader spends sampling resources.
`D3D10_COUNTER_VS_COMPUTATION_LIMITED` Percentage of time that a vertex shader spends doing computations.
`D3D10_COUNTER_GS_MEMORY_LIMITED` Percentage of time that a geometry shader spends sampling resources.
`D3D10_COUNTER_GS_COMPUTATION_LIMITED` Percentage of time that a geometry shader spends doing computations.
`D3D10_COUNTER_PS_MEMORY_LIMITED` Percentage of time that a pixel shader spends sampling resources.
`D3D10_COUNTER_PS_COMPUTATION_LIMITED` Percentage of time that a pixel shader spends doing computations.
`D3D10_COUNTER_POST_TRANSFORM_CACHE_HIT_RATE` Percentage of vertex data that was read from the vertex cache. For example, if 6 vertices were added to the cache and 3 of them were read from the cache, then the hit rate would be 0.5.

`D3D10_COUNTER_TEXTURE_CACHE_HIT_RATE`	Percentage of texel data that was read from the vertex cache. For example, if 6 texels were added to the cache and 3 of them were read from the cache, then the hit rate would be 0.5.
`D3D10_COUNTER_DEVICE_DEPENDENT_0`	Value: *0x40000000* Start of the device-dependent counters. See remarks.

Remarks

In addition to these performance counters, independent hardware vendors may define their own set of performance counters for their devices. The enum values for these counters would start after D3D10_COUNTER_DEVICE_DEPENDENT_0 and would be defined by those hardware vendors.

A device can support one or more of these performance counters, but it is not required to support any of them.

Requirements

Header	d3d10.h

See also

Core Enumerations

D3D10_COUNTER_TYPE enumeration (d3d10.h)

Article • 07/27/20222 minutes to read

Data type of a performance counter.

Syntax

C++

```
typedef enum D3D10_COUNTER_TYPE {
  D3D10_COUNTER_TYPE_FLOAT32 = 0,
  D3D10_COUNTER_TYPE_UINT16,
  D3D10_COUNTER_TYPE_UINT32,
  D3D10_COUNTER_TYPE_UINT64
} ;
```

Constants

`D3D10_COUNTER_TYPE_FLOAT32` Value: *0* 32-bit floating point.
`D3D10_COUNTER_TYPE_UINT16` 16-bit unsigned integer.
`D3D10_COUNTER_TYPE_UINT32` 32-bit unsigned integer.
`D3D10_COUNTER_TYPE_UINT64` 64-bit unsigned integer.

Remarks

These flags are an output parameter in ID3D10Device::CheckCounter.

Requirements

Header	d3d10.h

See also

Core Enumerations

D3D10_CREATE_DEVICE_FLAG enumeration (d3d10.h)

Article01/31/2022

Device creation flags.

Syntax

C++

```
typedef enum D3D10_CREATE_DEVICE_FLAG {
  D3D10_CREATE_DEVICE_SINGLETHREADED = 0x1,
  D3D10_CREATE_DEVICE_DEBUG = 0x2,
  D3D10_CREATE_DEVICE_SWITCH_TO_REF = 0x4,
  D3D10_CREATE_DEVICE_PREVENT_INTERNAL_THREADING_OPTIMIZATIONS = 0x8,
  D3D10_CREATE_DEVICE_ALLOW_NULL_FROM_MAP = 0x10,
  D3D10_CREATE_DEVICE_BGRA_SUPPORT = 0x20,
  D3D10_CREATE_DEVICE_PREVENT_ALTERING_LAYER_SETTINGS_FROM_REGISTRY = 0x80,
  D3D10_CREATE_DEVICE_STRICT_VALIDATION = 0x200,
  D3D10_CREATE_DEVICE_DEBUGGABLE = 0x400
} ;
```

Constants

`D3D10_CREATE_DEVICE_SINGLETHREADED` Value: *0x1* Use this flag if an application will only be calling D3D10 from a single thread. If this flag is not specified, the default behavior of D3D10 is to enter a lock during each API call to prevent multiple threads altering internal state. By using this flag no locks will be taken which can slightly increase performance, but could result in undefine behavior if D3D10 is called from multiple threads.
`D3D10_CREATE_DEVICE_DEBUG` Value: *0x2* Create a device that supports the debug layer.
`D3D10_CREATE_DEVICE_SWITCH_TO_REF` Value: *0x4* Create both a software (REF) and hardware (HAL) version of the device simultaneously, which allows an application to switch to a reference device to enable debugging. See ID3D10SwitchToRef Interface for more information.

`D3D10_CREATE_DEVICE_PREVENT_INTERNAL_THREADING_OPTIMIZATIONS`
Value: *0x8*
Prevents multiple threads from being created. When this flag is used with a WARP device, no additional threads will be created by WARP and all
rasterization will occur on the calling thread. This flag is not recommended for general use. See remarks.

`D3D10_CREATE_DEVICE_ALLOW_NULL_FROM_MAP`
Value: *0x10*
Return a **NULL** pointer instead of triggering an exception on memory exhaustion during invocations to Map.
Without this flag an exception will be raised on memory exhaustion. Only valid on Windows 7.

`D3D10_CREATE_DEVICE_BGRA_SUPPORT`
Value: *0x20*
Causes device creation to fail if BGRA support is not available.

BGRA support enables the following formats.

- DXGI_FORMAT_B8G8R8A8_TYPELESS
- DXGI_FORMAT_B8G8R8A8_UNORM
- DXGI_FORMAT_B8G8R8A8_UNORM_SRGB
- DXGI_FORMAT_B8G8R8X8_TYPELESS
- DXGI_FORMAT_B8G8R8X8_UNORM
- DXGI_FORMAT_B8G8R8X8_UNORM_SRGB

D3D10_CREATE_DEVICE_BGRA_SUPPORT is only relevant when a device is created with D3D10CreateDevice1
or D3D10CreateDeviceAndSwapChain1 using the **D3D10_FEATURE_LEVEL_10_0** or **D3D10_FEATURE_LEVEL_10_1**
feature levels, the flag will be ignored when a device is created with other feature levels.

Note that BGRA support may be present even if the application didn't specify D3D10_CREATE_DEVICE_BGRA_SUPPORT.
The flag merely causes device creation to fail if BGRA support isn't available.

D3D10_CREATE_DEVICE_BGRA_SUPPORT is only valid on Windows 7, Windows Server 2008 R2, and updated Windows Vista (KB971644) systems.

`D3D10_CREATE_DEVICE_PREVENT_ALTERING_LAYER_SETTINGS_FROM_REGISTRY` Value: *0x80* Causes the Direct3D runtime to ignore registry settings that turn on the debug layer. You can turn on the debug layer by using the DirectX Control Panel that was included as part of the DirectX SDK. We shipped the last version of the DirectX SDK in June 2010; you can download it from the Microsoft Download Center. You can set this flag in your app, typically in release builds only, to prevent end users from using the DirectX Control Panel to monitor how the app uses Direct3D. **Note** You can also set this flag in your app to prevent Direct3D debugging tools, such as Visual Studio Ultimate 2012, from hooking your app. **Windows 8.1:** This flag doesn't prevent Visual Studio 2013 and later running on Windows 8.1 and later from hooking your app. But, this flag still prevents Visual Studio 2013 and later running on Windows 8 and earlier from hooking your app. **Direct3D 11:** This value is not supported until Direct3D 11.1.	
`D3D10_CREATE_DEVICE_STRICT_VALIDATION` Value: *0x200* Reserved. This flag is currently not supported. Do not use.	
`D3D10_CREATE_DEVICE_DEBUGGABLE` Value: *0x400* Causes the device and driver to keep information that you can use for shader debugging. The exact impact from this flag will vary from driver to driver. To use this flag, you must have D3D11_1SDKLayers.dll installed; otherwise, device creation fails. The created device supports the debug layer. To get D3D11_1SDKLayers.dll, you must install the SDK for Windows 8. **Direct3D 11:** This value is not supported until Direct3D 11.1.	

Remarks

Device creation flags are used by D3D10CreateDevice and D3D10CreateDeviceAndSwapChain.

An application might dynamically create (and destroy) threads to improve performance especially on a machine with multiple CPU cores. There may be cases, however, when an application needs to prevent extra threads from being created. This can happen when you want to simplify debugging, profile code or develop a tool for instance. For these cases, use D3D10_CREATE_DEVICE_PREVENT_INTERNAL_THREADING_OPTIMIZATIONS

to request that the runtime and video driver not create any additional threads that might interfere with the application.

Requirements

Header	d3d10.h

See also

Core Enumerations

D3D10_CULL_MODE enumeration (d3d10.h)

Article07/27/2022

Indicates triangles facing a particular direction are not drawn.

Syntax

C++

```
typedef enum D3D10_CULL_MODE {
  D3D10_CULL_NONE = 1,
  D3D10_CULL_FRONT = 2,
  D3D10_CULL_BACK = 3
} ;
```

Constants

`D3D10_CULL_NONE` Value: *1* Always draw all triangles.
`D3D10_CULL_FRONT` Value: *2* Do not draw triangles that are front-facing.
`D3D10_CULL_BACK` Value: *3* Do not draw triangles that are back-facing.

Remarks

This enumeration is part of a rasterizer-state object description (see D3D10_RASTERIZER_DESC).

Requirements

Header	d3d10.h

See also

Core Enumerations

D3D10_DEPTH_WRITE_MASK enumeration (d3d10.h)

Article • 07/27/20222 minutes to read

Identify the portion of a depth-stencil buffer for writing depth data.

Syntax

C++

```
typedef enum D3D10_DEPTH_WRITE_MASK {
  D3D10_DEPTH_WRITE_MASK_ZERO = 0,
  D3D10_DEPTH_WRITE_MASK_ALL = 1
} ;
```

Constants

`D3D10_DEPTH_WRITE_MASK_ZERO` Value: *0* Turn off writes to the depth-stencil buffer.
`D3D10_DEPTH_WRITE_MASK_ALL` Value: *1* Turn on writes to the depth-stencil buffer.

Requirements

Header	d3d10.h

See also

Core Enumerations

D3D10_FEATURE_LEVEL1 enumeration (d3d10_1.h)

Article08/23/2022

The version of hardware acceleration requested.

Syntax

C++

```
typedef enum D3D10_FEATURE_LEVEL1 {
  D3D10_FEATURE_LEVEL_10_0 = 0xa000,
  D3D10_FEATURE_LEVEL_10_1 = 0xa100,
  D3D10_FEATURE_LEVEL_9_1 = 0x9100,
  D3D10_FEATURE_LEVEL_9_2 = 0x9200,
  D3D10_FEATURE_LEVEL_9_3 = 0x9300
} ;
```

Constants

`D3D10_FEATURE_LEVEL_10_0` Value: *0xa000* The hardware supports Direct3D 10.0 features.
`D3D10_FEATURE_LEVEL_10_1` Value: *0xa100* The hardware supports Direct3D 10.1 features.
`D3D10_FEATURE_LEVEL_9_1` Value: *0x9100* The hardware supports 9.1 feature level.
`D3D10_FEATURE_LEVEL_9_2` Value: *0x9200* The hardware supports 9.2 feature level.
`D3D10_FEATURE_LEVEL_9_3` Value: *0x9300* The hardware supports 9.3 feature level.

Remarks

Use this enumeration when creating a device with D3D10CreateDevice1 or D3D10CreateDeviceAndSwapChain1.

Note that 10level9 feature levels 9_1, 9_2, and 9_3 are only available with the Direct3D 11 runtime (Windows 7, Windows Server 2008 R2, updated Windows Vista with Service Pack 2 (SP2) [KB 971644], and updated Windows Server 2008 [KB 971512]).

For information about limitations creating nonhardware-type devices on certain feature levels, see Limitations Creating WARP and Reference Devices.

For an overview of the capabilities of each feature level, see Overview For Each Feature Level.

Requirements

Header	d3d10_1.h

See also

Core Enumerations

Hardware Support for Direct3D 10 Formats

Hardware Support for Direct3D 10.1 Formats

D3D10_DEVICE_STATE_TYPES enumeration (d3d10effect.h)

Article • 01/31/20222 minutes to read

Effect device-state types.

Syntax

C++

```
typedef enum _D3D10_DEVICE_STATE_TYPES {
  D3D10_DST_SO_BUFFERS = 1,
  D3D10_DST_OM_RENDER_TARGETS,
  D3D10_DST_OM_DEPTH_STENCIL_STATE,
  D3D10_DST_OM_BLEND_STATE,
  D3D10_DST_VS,
  D3D10_DST_VS_SAMPLERS,
  D3D10_DST_VS_SHADER_RESOURCES,
  D3D10_DST_VS_CONSTANT_BUFFERS,
  D3D10_DST_GS,
  D3D10_DST_GS_SAMPLERS,
  D3D10_DST_GS_SHADER_RESOURCES,
  D3D10_DST_GS_CONSTANT_BUFFERS,
  D3D10_DST_PS,
  D3D10_DST_PS_SAMPLERS,
  D3D10_DST_PS_SHADER_RESOURCES,
  D3D10_DST_PS_CONSTANT_BUFFERS,
  D3D10_DST_IA_VERTEX_BUFFERS,
  D3D10_DST_IA_INDEX_BUFFER,
  D3D10_DST_IA_INPUT_LAYOUT,
  D3D10_DST_IA_PRIMITIVE_TOPOLOGY,
  D3D10_DST_RS_VIEWPORTS,
  D3D10_DST_RS_SCISSOR_RECTS,
  D3D10_DST_RS_RASTERIZER_STATE,
  D3D10_DST_PREDICATION
} D3D10_DEVICE_STATE_TYPES;
```

Constants

`D3D10_DST_SO_BUFFERS` Value: *1* Stream-output buffer.

`D3D10_DST_OM_RENDER_TARGETS` Render target.
`D3D10_DST_OM_DEPTH_STENCIL_STATE` Depth-stencil state.
`D3D10_DST_OM_BLEND_STATE` Blend state.
`D3D10_DST_VS` Vertex shader.
`D3D10_DST_VS_SAMPLERS` Vertex shader sampler.
`D3D10_DST_VS_SHADER_RESOURCES` Vertex shader resource.
`D3D10_DST_VS_CONSTANT_BUFFERS` Vertex shader constant buffer.
`D3D10_DST_GS` Geometry shader.
`D3D10_DST_GS_SAMPLERS` Geometry shader sampler.
`D3D10_DST_GS_SHADER_RESOURCES` Geometry shader resource.
`D3D10_DST_GS_CONSTANT_BUFFERS` Geometry shader constant buffer.
`D3D10_DST_PS` Pixel shader.
`D3D10_DST_PS_SAMPLERS` Pixel shader sampler.
`D3D10_DST_PS_SHADER_RESOURCES` Pixel shader resource.
`D3D10_DST_PS_CONSTANT_BUFFERS` Pixel shader constant buffer.
`D3D10_DST_IA_VERTEX_BUFFERS` Input-assembler vertex buffer.

`D3D10_DST_IA_INDEX_BUFFER`	Input-assembler index buffer.
`D3D10_DST_IA_INPUT_LAYOUT`	Input-assembler input layout.
`D3D10_DST_IA_PRIMITIVE_TOPOLOGY`	Input-assembler primitive topology.
`D3D10_DST_RS_VIEWPORTS`	Viewport.
`D3D10_DST_RS_SCISSOR_RECTS`	Scissor rectangle.
`D3D10_DST_RS_RASTERIZER_STATE`	Rasterizer state.
`D3D10_DST_PREDICATION`	Predication state.

Remarks

This enumeration is used by D3D10StateBlockMaskDisableCapture, D3D10StateBlockMaskEnableCapture, and D3D10StateBlockMaskGetSetting.

Requirements

Header	d3d10effect.h

See also

Core Enumerations

D3D10_DRIVER_TYPE enumeration (d3d10misc.h)

Article01/31/2022

The device-driver type.

Syntax

C++

```
typedef enum D3D10_DRIVER_TYPE {
  D3D10_DRIVER_TYPE_HARDWARE = 0,
  D3D10_DRIVER_TYPE_REFERENCE = 1,
  D3D10_DRIVER_TYPE_NULL = 2,
  D3D10_DRIVER_TYPE_SOFTWARE = 3,
  D3D10_DRIVER_TYPE_WARP = 5
} ;
```

Constants

`D3D10_DRIVER_TYPE_HARDWARE` Value: *0* A hardware device; commonly called a HAL device.
`D3D10_DRIVER_TYPE_REFERENCE` Value: *1* A reference device; commonly called a REF device.
`D3D10_DRIVER_TYPE_NULL` Value: *2* A NULL device; which is a reference device without render capability.
`D3D10_DRIVER_TYPE_SOFTWARE` Value: *3* Reserved for later use.

`D3D10_DRIVER_TYPE_WARP` Value: *5* A WARP driver, which is a high-performance software rasterizer. The rasterizer supports feature level 9_1 through level 10.1 with a high performance software implementation when hardware is not available. For more information about using a WARP driver, see Windows Advanced Rasterization Platform (WARP) In-Depth Guide. Note that WARP is only available with the DirectX 11 Runtime (Windows 7, Windows Server 2008 R2, updated Windows Vista [KB971644]).	

Remarks

The device-driver type needs to be specified when the device is created (using D3D10CreateDevice or D3D10CreateDeviceAndSwapChain).

For information about limitations creating nonhardware-type devices on certain feature levels, see Limitations Creating WARP and Reference Devices.

Requirements

Header	d3d10misc.h

See also

Core Enumerations

D3D10_FILL_MODE enumeration (d3d10.h)

Article • 07/27/20222 minutes to read

Determines the fill mode to use when rendering triangles.

Syntax

C++

```
typedef enum D3D10_FILL_MODE {
  D3D10_FILL_WIREFRAME = 2,
  D3D10_FILL_SOLID = 3
} ;
```

Constants

`D3D10_FILL_WIREFRAME` Value: *2* Draw lines connecting the vertices. Adjacent vertices are not drawn.
`D3D10_FILL_SOLID` Value: *3* Fill the triangles formed by the vertices. Adjacent vertices are not drawn.

Remarks

This enumeration is part of a rasterizer-state object description (see D3D10_RASTERIZER_DESC).

Requirements

Header	d3d10.h

See also

Core Enumerations

D3D10_FILTER enumeration (d3d10.h)

Article • 07/27/20222 minutes to read

Filtering options during texture sampling.

Syntax

C++

```
typedef enum D3D10_FILTER {
  D3D10_FILTER_MIN_MAG_MIP_POINT = 0,
  D3D10_FILTER_MIN_MAG_POINT_MIP_LINEAR = 0x1,
  D3D10_FILTER_MIN_POINT_MAG_LINEAR_MIP_POINT = 0x4,
  D3D10_FILTER_MIN_POINT_MAG_MIP_LINEAR = 0x5,
  D3D10_FILTER_MIN_LINEAR_MAG_MIP_POINT = 0x10,
  D3D10_FILTER_MIN_LINEAR_MAG_POINT_MIP_LINEAR = 0x11,
  D3D10_FILTER_MIN_MAG_LINEAR_MIP_POINT = 0x14,
  D3D10_FILTER_MIN_MAG_MIP_LINEAR = 0x15,
  D3D10_FILTER_ANISOTROPIC = 0x55,
  D3D10_FILTER_COMPARISON_MIN_MAG_MIP_POINT = 0x80,
  D3D10_FILTER_COMPARISON_MIN_MAG_POINT_MIP_LINEAR = 0x81,
  D3D10_FILTER_COMPARISON_MIN_POINT_MAG_LINEAR_MIP_POINT = 0x84,
  D3D10_FILTER_COMPARISON_MIN_POINT_MAG_MIP_LINEAR = 0x85,
  D3D10_FILTER_COMPARISON_MIN_LINEAR_MAG_MIP_POINT = 0x90,
  D3D10_FILTER_COMPARISON_MIN_LINEAR_MAG_POINT_MIP_LINEAR = 0x91,
  D3D10_FILTER_COMPARISON_MIN_MAG_LINEAR_MIP_POINT = 0x94,
  D3D10_FILTER_COMPARISON_MIN_MAG_MIP_LINEAR = 0x95,
  D3D10_FILTER_COMPARISON_ANISOTROPIC = 0xd5,
  D3D10_FILTER_TEXT_1BIT = 0x80000000
} ;
```

Constants

`D3D10_FILTER_MIN_MAG_MIP_POINT` Value: *0* Use point sampling for minification, magnification, and mip-level sampling.
`D3D10_FILTER_MIN_MAG_POINT_MIP_LINEAR` Value: *0x1* Use point sampling for minification and magnification; use linear interpolation for mip-level sampling.

`D3D10_FILTER_MIN_POINT_MAG_LINEAR_MIP_POINT` Value: *0x4* Use point sampling for minification; use linear interpolation for magnification; use point sampling for mip-level sampling.
`D3D10_FILTER_MIN_POINT_MAG_MIP_LINEAR` Value: *0x5* Use point sampling for minification; use linear interpolation for magnification and mip-level sampling.
`D3D10_FILTER_MIN_LINEAR_MAG_MIP_POINT` Value: *0x10* Use linear interpolation for minification; use point sampling for magnification and mip-level sampling.
`D3D10_FILTER_MIN_LINEAR_MAG_POINT_MIP_LINEAR` Value: *0x11* Use linear interpolation for minification; use point sampling for magnification; use linear interpolation for mip-level sampling.
`D3D10_FILTER_MIN_MAG_LINEAR_MIP_POINT` Value: *0x14* Use linear interpolation for minification and magnification; use point sampling for mip-level sampling.
`D3D10_FILTER_MIN_MAG_MIP_LINEAR` Value: *0x15* Use linear interpolation for minification, magnification, and mip-level sampling.
`D3D10_FILTER_ANISOTROPIC` Value: *0x55* Use anisotropic interpolation for minification, magnification, and mip-level sampling.
`D3D10_FILTER_COMPARISON_MIN_MAG_MIP_POINT` Value: *0x80* Use point sampling for minification, magnification, and mip-level sampling. Compare the result to the comparison value.
`D3D10_FILTER_COMPARISON_MIN_MAG_POINT_MIP_LINEAR` Value: *0x81* Use point sampling for minification and magnification; use linear interpolation for mip-level sampling. Compare the result to the comparison value.
`D3D10_FILTER_COMPARISON_MIN_POINT_MAG_LINEAR_MIP_POINT` Value: *0x84* Use point sampling for minification; use linear interpolation for magnification; use point sampling for mip-level sampling. Compare the result to the comparison value.

`D3D10_FILTER_COMPARISON_MIN_POINT_MAG_MIP_LINEAR` Value: *0x85* Use point sampling for minification; use linear interpolation for magnification and mip-level sampling. Compare the result to the comparison value.
`D3D10_FILTER_COMPARISON_MIN_LINEAR_MAG_MIP_POINT` Value: *0x90* Use linear interpolation for minification; use point sampling for magnification and mip-level sampling. Compare the result to the comparison value.
`D3D10_FILTER_COMPARISON_MIN_LINEAR_MAG_POINT_MIP_LINEAR` Value: *0x91* Use linear interpolation for minification; use point sampling for magnification; use linear interpolation for mip-level sampling. Compare the result to the comparison value.
`D3D10_FILTER_COMPARISON_MIN_MAG_LINEAR_MIP_POINT` Value: *0x94* Use linear interpolation for minification and magnification; use point sampling for mip-level sampling. Compare the result to the comparison value.
`D3D10_FILTER_COMPARISON_MIN_MAG_MIP_LINEAR` Value: *0x95* Use linear interpolation for minification, magnification, and mip-level sampling. Compare the result to the comparison value.
`D3D10_FILTER_COMPARISON_ANISOTROPIC` Value: *0xd5* Use anisotropic interpolation for minification, magnification, and mip-level sampling. Compare the result to the comparison value.
`D3D10_FILTER_TEXT_1BIT` Value: *0x80000000* For use in pixel shaders with textures that have the R1_UNORM format.

Remarks

During texture sampling, one or more texels are read and combined (this is calling filtering) to produce a single value. Point sampling reads a single texel while linear sampling reads two texels (endpoints) and linearly interpolates a third value between the endpoints.

HLSL texture-sampling functions also support comparison filtering during texture sampling. Comparison filtering compares each sampled texel against a comparison value. The boolean result is blended the same way that normal texture filtering is blended.

You can use HLSL intrinsic texture-sampling functions that implement texture filtering only or companion functions that use texture filtering with comparison filtering.

Texture Sampling Function	Texture Sampling Function with Comparison Filtering
sample	samplecmp or samplecmplevelzero

Comparison filters only work with textures that have the following formats: R32_FLOAT_X8X24_TYPELESS, R32_FLOAT, R24_UNORM_X8_TYPELESS, R16_UNORM.

Requirements

Header	d3d10.h

See also

Core Enumerations

D3D10_FILTER_TYPE enumeration (d3d10.h)

Article07/27/2022

Types of magnification or minification sampler filters.

Syntax

C++

```
typedef enum D3D10_FILTER_TYPE {
  D3D10_FILTER_TYPE_POINT = 0,
  D3D10_FILTER_TYPE_LINEAR = 1
} ;
```

Constants

`D3D10_FILTER_TYPE_POINT` Value: *0* Point filtering used as a texture magnification or minification filter. The texel with coordinates nearest to the desired pixel value is used. The texture filter to be used between mipmap levels is nearest-point mipmap filtering. The rasterizer uses the color from the texel of the nearest mipmap texture.
`D3D10_FILTER_TYPE_LINEAR` Value: *1* Bilinear interpolation filtering used as a texture magnification or minification filter. A weighted average of a 2 x 2 area of texels surrounding the desired pixel is used. The texture filter to use between mipmap levels is trilinear mipmap interpolation. The rasterizer linearly interpolates pixel color, using the texels of the two nearest mipmap textures.

Requirements

Header	d3d10.h

See also

D3D10_FORMAT_SUPPORT enumeration (d3d10.h)

Article • 01/31/20222 minutes to read

Which resources are supported for a given format and given device (see ID3D10Device::CheckFormatSupport).

Syntax

C++

```
typedef enum D3D10_FORMAT_SUPPORT {
  D3D10_FORMAT_SUPPORT_BUFFER = 0x1,
  D3D10_FORMAT_SUPPORT_IA_VERTEX_BUFFER = 0x2,
  D3D10_FORMAT_SUPPORT_IA_INDEX_BUFFER = 0x4,
  D3D10_FORMAT_SUPPORT_SO_BUFFER = 0x8,
  D3D10_FORMAT_SUPPORT_TEXTURE1D = 0x10,
  D3D10_FORMAT_SUPPORT_TEXTURE2D = 0x20,
  D3D10_FORMAT_SUPPORT_TEXTURE3D = 0x40,
  D3D10_FORMAT_SUPPORT_TEXTURECUBE = 0x80,
  D3D10_FORMAT_SUPPORT_SHADER_LOAD = 0x100,
  D3D10_FORMAT_SUPPORT_SHADER_SAMPLE = 0x200,
  D3D10_FORMAT_SUPPORT_SHADER_SAMPLE_COMPARISON = 0x400,
  D3D10_FORMAT_SUPPORT_SHADER_SAMPLE_MONO_TEXT = 0x800,
  D3D10_FORMAT_SUPPORT_MIP = 0x1000,
  D3D10_FORMAT_SUPPORT_MIP_AUTOGEN = 0x2000,
  D3D10_FORMAT_SUPPORT_RENDER_TARGET = 0x4000,
  D3D10_FORMAT_SUPPORT_BLENDABLE = 0x8000,
  D3D10_FORMAT_SUPPORT_DEPTH_STENCIL = 0x10000,
  D3D10_FORMAT_SUPPORT_CPU_LOCKABLE = 0x20000,
  D3D10_FORMAT_SUPPORT_MULTISAMPLE_RESOLVE = 0x40000,
  D3D10_FORMAT_SUPPORT_DISPLAY = 0x80000,
  D3D10_FORMAT_SUPPORT_CAST_WITHIN_BIT_LAYOUT = 0x100000,
  D3D10_FORMAT_SUPPORT_MULTISAMPLE_RENDERTARGET = 0x200000,
  D3D10_FORMAT_SUPPORT_MULTISAMPLE_LOAD = 0x400000,
  D3D10_FORMAT_SUPPORT_SHADER_GATHER = 0x800000,
  D3D10_FORMAT_SUPPORT_BACK_BUFFER_CAST = 0x1000000
} ;
```

Constants

`D3D10_FORMAT_SUPPORT_BUFFER` Value: *0x1* Buffer resources supported.
`D3D10_FORMAT_SUPPORT_IA_VERTEX_BUFFER` Value: *0x2* Vertex buffers supported.
`D3D10_FORMAT_SUPPORT_IA_INDEX_BUFFER` Value: *0x4* Index buffers supported.
`D3D10_FORMAT_SUPPORT_SO_BUFFER` Value: *0x8* Streaming output buffers supported.
`D3D10_FORMAT_SUPPORT_TEXTURE1D` Value: *0x10* 1D texture resources supported.
`D3D10_FORMAT_SUPPORT_TEXTURE2D` Value: *0x20* 2D texture resources supported.
`D3D10_FORMAT_SUPPORT_TEXTURE3D` Value: *0x40* 3D texture resources supported.
`D3D10_FORMAT_SUPPORT_TEXTURECUBE` Value: *0x80* Cube texture resources supported.
`D3D10_FORMAT_SUPPORT_SHADER_LOAD` Value: *0x100* The intrinsic HLSL function load is supported.
`D3D10_FORMAT_SUPPORT_SHADER_SAMPLE` Value: *0x200* The intrinsic HLSL functions Sample supported.
`D3D10_FORMAT_SUPPORT_SHADER_SAMPLE_COMPARISON` Value: *0x400* The intrinsic HLSL functions SampleCmp and SampleCmpLevelZero are supported.
`D3D10_FORMAT_SUPPORT_SHADER_SAMPLE_MONO_TEXT` Value: *0x800* Reserved.

`D3D10_FORMAT_SUPPORT_MIP` Value: *0x1000* Mipmaps are supported.
`D3D10_FORMAT_SUPPORT_MIP_AUTOGEN` Value: *0x2000* Automatic generation of mipmaps is supported.
`D3D10_FORMAT_SUPPORT_RENDER_TARGET` Value: *0x4000* Rendertargets are supported.
`D3D10_FORMAT_SUPPORT_BLENDABLE` Value: *0x8000* Render target blend operations supported.
`D3D10_FORMAT_SUPPORT_DEPTH_STENCIL` Value: *0x10000* Depth stencils supported.
`D3D10_FORMAT_SUPPORT_CPU_LOCKABLE` Value: *0x20000* CPU locking supported.
`D3D10_FORMAT_SUPPORT_MULTISAMPLE_RESOLVE` Value: *0x40000* Multisampling resolution supported.
`D3D10_FORMAT_SUPPORT_DISPLAY` Value: *0x80000* Format can be displayed on screen.
`D3D10_FORMAT_SUPPORT_CAST_WITHIN_BIT_LAYOUT` Value: *0x100000* Format cannot be cast to another format.
`D3D10_FORMAT_SUPPORT_MULTISAMPLE_RENDERTARGET` Value: *0x200000* Format can be used as a multisampled rendertarget.
`D3D10_FORMAT_SUPPORT_MULTISAMPLE_LOAD` Value: *0x400000* Format can be used as a multisampled texture and read into a shader with the load function.
`D3D10_FORMAT_SUPPORT_SHADER_GATHER` Value: *0x800000* Format can be used with the gather function. This value is available in DirectX 10.1 or higher.

`D3D10_FORMAT_SUPPORT_BACK_BUFFER_CAST` Value: *0x1000000*

Requirements

Header	d3d10.h

See also

Core Enumerations

D3D10_INPUT_CLASSIFICATION enumeration (d3d10.h)

Article • 07/27/20222 minutes to read

Type of data contained in an input slot.

Syntax

C++

```
typedef enum D3D10_INPUT_CLASSIFICATION {
  D3D10_INPUT_PER_VERTEX_DATA = 0,
  D3D10_INPUT_PER_INSTANCE_DATA = 1
} ;
```

Constants

`D3D10_INPUT_PER_VERTEX_DATA` Value: *0* Input data is per-vertex data.
`D3D10_INPUT_PER_INSTANCE_DATA` Value: *1* Input data is per-instance data.

Remarks

Use these values to specify the type of data for a particular input element (see D3D10_INPUT_ELEMENT_DESC) of an input-layout object.

Requirements

Header	d3d10.h

See also

Core Enumerations

D3D10_MESSAGE_CATEGORY enumeration (d3d10sdklayers.h)

Article • 07/27/20222 minutes to read

Categories of debug messages. This will identify the category of a message when retrieving a message with ID3D10InfoQueue::GetMessage and when adding a message with ID3D10InfoQueue::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.

Syntax

C++

```
typedef enum D3D10_MESSAGE_CATEGORY {
  D3D10_MESSAGE_CATEGORY_APPLICATION_DEFINED = 0,
  D3D10_MESSAGE_CATEGORY_MISCELLANEOUS,
  D3D10_MESSAGE_CATEGORY_INITIALIZATION,
  D3D10_MESSAGE_CATEGORY_CLEANUP,
  D3D10_MESSAGE_CATEGORY_COMPILATION,
  D3D10_MESSAGE_CATEGORY_STATE_CREATION,
  D3D10_MESSAGE_CATEGORY_STATE_SETTING,
  D3D10_MESSAGE_CATEGORY_STATE_GETTING,
  D3D10_MESSAGE_CATEGORY_RESOURCE_MANIPULATION,
  D3D10_MESSAGE_CATEGORY_EXECUTION,
  D3D10_MESSAGE_CATEGORY_SHADER
} ;
```

Constants

`D3D10_MESSAGE_CATEGORY_APPLICATION_DEFINED` Value: *0* User defined message. See ID3D10InfoQueue::AddMessage.
`D3D10_MESSAGE_CATEGORY_MISCELLANEOUS`
`D3D10_MESSAGE_CATEGORY_INITIALIZATION`
`D3D10_MESSAGE_CATEGORY_CLEANUP`
`D3D10_MESSAGE_CATEGORY_COMPILATION`

`D3D10_MESSAGE_CATEGORY_STATE_CREATION`
`D3D10_MESSAGE_CATEGORY_STATE_SETTING`
`D3D10_MESSAGE_CATEGORY_STATE_GETTING`
`D3D10_MESSAGE_CATEGORY_RESOURCE_MANIPULATION`
`D3D10_MESSAGE_CATEGORY_EXECUTION`
`D3D10_MESSAGE_CATEGORY_SHADER`

Remarks

This is part of the Information Queue feature. See ID3D10InfoQueue Interface.

Requirements

Header	d3d10sdklayers.h (include D3D10.h)

See also

Core Enumerations

D3D10_MESSAGE_ID enumeration (d3d10sdklayers.h)

Article01/31/2022

Debug messages for setting up an info-queue filter (see D3D10_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 ID3D10InfoQueue::GetMessage or ID3D10InfoQueue::AddMessage.

Syntax

C++

```
typedef enum D3D10_MESSAGE_ID {
  D3D10_MESSAGE_ID_UNKNOWN = 0,
  D3D10_MESSAGE_ID_DEVICE_IASETVERTEXBUFFERS_HAZARD,
  D3D10_MESSAGE_ID_DEVICE_IASETINDEXBUFFER_HAZARD,
  D3D10_MESSAGE_ID_DEVICE_VSSETSHADERRESOURCES_HAZARD,
  D3D10_MESSAGE_ID_DEVICE_VSSETCONSTANTBUFFERS_HAZARD,
  D3D10_MESSAGE_ID_DEVICE_GSSETSHADERRESOURCES_HAZARD,
  D3D10_MESSAGE_ID_DEVICE_GSSETCONSTANTBUFFERS_HAZARD,
  D3D10_MESSAGE_ID_DEVICE_PSSETSHADERRESOURCES_HAZARD,
  D3D10_MESSAGE_ID_DEVICE_PSSETCONSTANTBUFFERS_HAZARD,
  D3D10_MESSAGE_ID_DEVICE_OMSETRENDERTARGETS_HAZARD,
  D3D10_MESSAGE_ID_DEVICE_SOSETTARGETS_HAZARD,
  D3D10_MESSAGE_ID_STRING_FROM_APPLICATION,
  D3D10_MESSAGE_ID_CORRUPTED_THIS,
  D3D10_MESSAGE_ID_CORRUPTED_PARAMETER1,
  D3D10_MESSAGE_ID_CORRUPTED_PARAMETER2,
  D3D10_MESSAGE_ID_CORRUPTED_PARAMETER3,
  D3D10_MESSAGE_ID_CORRUPTED_PARAMETER4,
  D3D10_MESSAGE_ID_CORRUPTED_PARAMETER5,
  D3D10_MESSAGE_ID_CORRUPTED_PARAMETER6,
  D3D10_MESSAGE_ID_CORRUPTED_PARAMETER7,
  D3D10_MESSAGE_ID_CORRUPTED_PARAMETER8,
  D3D10_MESSAGE_ID_CORRUPTED_PARAMETER9,
  D3D10_MESSAGE_ID_CORRUPTED_PARAMETER10,
  D3D10_MESSAGE_ID_CORRUPTED_PARAMETER11,
  D3D10_MESSAGE_ID_CORRUPTED_PARAMETER12,
  D3D10_MESSAGE_ID_CORRUPTED_PARAMETER13,
  D3D10_MESSAGE_ID_CORRUPTED_PARAMETER14,
  D3D10_MESSAGE_ID_CORRUPTED_PARAMETER15,
  D3D10_MESSAGE_ID_CORRUPTED_MULTITHREADING,
  D3D10_MESSAGE_ID_MESSAGE_REPORTING_OUTOFMEMORY,
  D3D10_MESSAGE_ID_IASETINPUTLAYOUT_UNBINDDELETINGOBJECT,
  D3D10_MESSAGE_ID_IASETVERTEXBUFFERS_UNBINDDELETINGOBJECT,
  D3D10_MESSAGE_ID_IASETINDEXBUFFER_UNBINDDELETINGOBJECT,
  D3D10_MESSAGE_ID_VSSETSHADER_UNBINDDELETINGOBJECT,
```

```
    D3D10_MESSAGE_ID_VSSETSHADERRESOURCES_UNBINDDELETINGOBJECT,
    D3D10_MESSAGE_ID_VSSETCONSTANTBUFFERS_UNBINDDELETINGOBJECT,
    D3D10_MESSAGE_ID_VSSETSAMPLERS_UNBINDDELETINGOBJECT,
    D3D10_MESSAGE_ID_GSSETSHADER_UNBINDDELETINGOBJECT,
    D3D10_MESSAGE_ID_GSSETSHADERRESOURCES_UNBINDDELETINGOBJECT,
    D3D10_MESSAGE_ID_GSSETCONSTANTBUFFERS_UNBINDDELETINGOBJECT,
    D3D10_MESSAGE_ID_GSSETSAMPLERS_UNBINDDELETINGOBJECT,
    D3D10_MESSAGE_ID_SOSETTARGETS_UNBINDDELETINGOBJECT,
    D3D10_MESSAGE_ID_PSSETSHADER_UNBINDDELETINGOBJECT,
    D3D10_MESSAGE_ID_PSSETSHADERRESOURCES_UNBINDDELETINGOBJECT,
    D3D10_MESSAGE_ID_PSSETCONSTANTBUFFERS_UNBINDDELETINGOBJECT,
    D3D10_MESSAGE_ID_PSSETSAMPLERS_UNBINDDELETINGOBJECT,
    D3D10_MESSAGE_ID_RSSETSTATE_UNBINDDELETINGOBJECT,
    D3D10_MESSAGE_ID_OMSETBLENDSTATE_UNBINDDELETINGOBJECT,
    D3D10_MESSAGE_ID_OMSETDEPTHSTENCILSTATE_UNBINDDELETINGOBJECT,
    D3D10_MESSAGE_ID_OMSETRENDERTARGETS_UNBINDDELETINGOBJECT,
    D3D10_MESSAGE_ID_SETPREDICATION_UNBINDDELETINGOBJECT,
    D3D10_MESSAGE_ID_GETPRIVATEDATA_MOREDATA,
    D3D10_MESSAGE_ID_SETPRIVATEDATA_INVALIDFREEDATA,
    D3D10_MESSAGE_ID_SETPRIVATEDATA_INVALIDIUNKNOWN,
    D3D10_MESSAGE_ID_SETPRIVATEDATA_INVALIDFLAGS,
    D3D10_MESSAGE_ID_SETPRIVATEDATA_CHANGINGPARAMS,
    D3D10_MESSAGE_ID_SETPRIVATEDATA_OUTOFMEMORY,
    D3D10_MESSAGE_ID_CREATEBUFFER_UNRECOGNIZEDFORMAT,
    D3D10_MESSAGE_ID_CREATEBUFFER_INVALIDSAMPLES,
    D3D10_MESSAGE_ID_CREATEBUFFER_UNRECOGNIZEDUSAGE,
    D3D10_MESSAGE_ID_CREATEBUFFER_UNRECOGNIZEDBINDFLAGS,
    D3D10_MESSAGE_ID_CREATEBUFFER_UNRECOGNIZEDCPUACCESSFLAGS,
    D3D10_MESSAGE_ID_CREATEBUFFER_UNRECOGNIZEDMISCFLAGS,
    D3D10_MESSAGE_ID_CREATEBUFFER_INVALIDCPUACCESSFLAGS,
    D3D10_MESSAGE_ID_CREATEBUFFER_INVALIDBINDFLAGS,
    D3D10_MESSAGE_ID_CREATEBUFFER_INVALIDINITIALDATA,
    D3D10_MESSAGE_ID_CREATEBUFFER_INVALIDDIMENSIONS,
    D3D10_MESSAGE_ID_CREATEBUFFER_INVALIDMIPLEVELS,
    D3D10_MESSAGE_ID_CREATEBUFFER_INVALIDMISCFLAGS,
    D3D10_MESSAGE_ID_CREATEBUFFER_INVALIDARG_RETURN,
    D3D10_MESSAGE_ID_CREATEBUFFER_OUTOFMEMORY_RETURN,
    D3D10_MESSAGE_ID_CREATEBUFFER_NULLDESC,
    D3D10_MESSAGE_ID_CREATEBUFFER_INVALIDCONSTANTBUFFERBINDINGS,
    D3D10_MESSAGE_ID_CREATEBUFFER_LARGEALLOCATION,
    D3D10_MESSAGE_ID_CREATETEXTURE1D_UNRECOGNIZEDFORMAT,
    D3D10_MESSAGE_ID_CREATETEXTURE1D_UNSUPPORTEDFORMAT,
    D3D10_MESSAGE_ID_CREATETEXTURE1D_INVALIDSAMPLES,
    D3D10_MESSAGE_ID_CREATETEXTURE1D_UNRECOGNIZEDUSAGE,
    D3D10_MESSAGE_ID_CREATETEXTURE1D_UNRECOGNIZEDBINDFLAGS,
    D3D10_MESSAGE_ID_CREATETEXTURE1D_UNRECOGNIZEDCPUACCESSFLAGS,
    D3D10_MESSAGE_ID_CREATETEXTURE1D_UNRECOGNIZEDMISCFLAGS,
    D3D10_MESSAGE_ID_CREATETEXTURE1D_INVALIDCPUACCESSFLAGS,
    D3D10_MESSAGE_ID_CREATETEXTURE1D_INVALIDBINDFLAGS,
    D3D10_MESSAGE_ID_CREATETEXTURE1D_INVALIDINITIALDATA,
    D3D10_MESSAGE_ID_CREATETEXTURE1D_INVALIDDIMENSIONS,
    D3D10_MESSAGE_ID_CREATETEXTURE1D_INVALIDMIPLEVELS,
    D3D10_MESSAGE_ID_CREATETEXTURE1D_INVALIDMISCFLAGS,
    D3D10_MESSAGE_ID_CREATETEXTURE1D_INVALIDARG_RETURN,
    D3D10_MESSAGE_ID_CREATETEXTURE1D_OUTOFMEMORY_RETURN,
```

```
    D3D10_MESSAGE_ID_CREATETEXTURE1D_NULLDESC,
    D3D10_MESSAGE_ID_CREATETEXTURE1D_LARGEALLOCATION,
    D3D10_MESSAGE_ID_CREATETEXTURE2D_UNRECOGNIZEDFORMAT,
    D3D10_MESSAGE_ID_CREATETEXTURE2D_UNSUPPORTEDFORMAT,
    D3D10_MESSAGE_ID_CREATETEXTURE2D_INVALIDSAMPLES,
    D3D10_MESSAGE_ID_CREATETEXTURE2D_UNRECOGNIZEDUSAGE,
    D3D10_MESSAGE_ID_CREATETEXTURE2D_UNRECOGNIZEDBINDFLAGS,
    D3D10_MESSAGE_ID_CREATETEXTURE2D_UNRECOGNIZEDCPUACCESSFLAGS,
    D3D10_MESSAGE_ID_CREATETEXTURE2D_UNRECOGNIZEDMISCFLAGS,
    D3D10_MESSAGE_ID_CREATETEXTURE2D_INVALIDCPUACCESSFLAGS,
    D3D10_MESSAGE_ID_CREATETEXTURE2D_INVALIDBINDFLAGS,
    D3D10_MESSAGE_ID_CREATETEXTURE2D_INVALIDINITIALDATA,
    D3D10_MESSAGE_ID_CREATETEXTURE2D_INVALIDDIMENSIONS,
    D3D10_MESSAGE_ID_CREATETEXTURE2D_INVALIDMIPLEVELS,
    D3D10_MESSAGE_ID_CREATETEXTURE2D_INVALIDMISCFLAGS,
    D3D10_MESSAGE_ID_CREATETEXTURE2D_INVALIDARG_RETURN,
    D3D10_MESSAGE_ID_CREATETEXTURE2D_OUTOFMEMORY_RETURN,
    D3D10_MESSAGE_ID_CREATETEXTURE2D_NULLDESC,
    D3D10_MESSAGE_ID_CREATETEXTURE2D_LARGEALLOCATION,
    D3D10_MESSAGE_ID_CREATETEXTURE3D_UNRECOGNIZEDFORMAT,
    D3D10_MESSAGE_ID_CREATETEXTURE3D_UNSUPPORTEDFORMAT,
    D3D10_MESSAGE_ID_CREATETEXTURE3D_INVALIDSAMPLES,
    D3D10_MESSAGE_ID_CREATETEXTURE3D_UNRECOGNIZEDUSAGE,
    D3D10_MESSAGE_ID_CREATETEXTURE3D_UNRECOGNIZEDBINDFLAGS,
    D3D10_MESSAGE_ID_CREATETEXTURE3D_UNRECOGNIZEDCPUACCESSFLAGS,
    D3D10_MESSAGE_ID_CREATETEXTURE3D_UNRECOGNIZEDMISCFLAGS,
    D3D10_MESSAGE_ID_CREATETEXTURE3D_INVALIDCPUACCESSFLAGS,
    D3D10_MESSAGE_ID_CREATETEXTURE3D_INVALIDBINDFLAGS,
    D3D10_MESSAGE_ID_CREATETEXTURE3D_INVALIDINITIALDATA,
    D3D10_MESSAGE_ID_CREATETEXTURE3D_INVALIDDIMENSIONS,
    D3D10_MESSAGE_ID_CREATETEXTURE3D_INVALIDMIPLEVELS,
    D3D10_MESSAGE_ID_CREATETEXTURE3D_INVALIDMISCFLAGS,
    D3D10_MESSAGE_ID_CREATETEXTURE3D_INVALIDARG_RETURN,
    D3D10_MESSAGE_ID_CREATETEXTURE3D_OUTOFMEMORY_RETURN,
    D3D10_MESSAGE_ID_CREATETEXTURE3D_NULLDESC,
    D3D10_MESSAGE_ID_CREATETEXTURE3D_LARGEALLOCATION,
    D3D10_MESSAGE_ID_CREATESHADERRESOURCEVIEW_UNRECOGNIZEDFORMAT,
    D3D10_MESSAGE_ID_CREATESHADERRESOURCEVIEW_INVALIDDESC,
    D3D10_MESSAGE_ID_CREATESHADERRESOURCEVIEW_INVALIDFORMAT,
    D3D10_MESSAGE_ID_CREATESHADERRESOURCEVIEW_INVALIDDIMENSIONS,
    D3D10_MESSAGE_ID_CREATESHADERRESOURCEVIEW_INVALIDRESOURCE,
    D3D10_MESSAGE_ID_CREATESHADERRESOURCEVIEW_TOOMANYOBJECTS,
    D3D10_MESSAGE_ID_CREATESHADERRESOURCEVIEW_INVALIDARG_RETURN,
    D3D10_MESSAGE_ID_CREATESHADERRESOURCEVIEW_OUTOFMEMORY_RETURN,
    D3D10_MESSAGE_ID_CREATERENDERTARGETVIEW_UNRECOGNIZEDFORMAT,
    D3D10_MESSAGE_ID_CREATERENDERTARGETVIEW_UNSUPPORTEDFORMAT,
    D3D10_MESSAGE_ID_CREATERENDERTARGETVIEW_INVALIDDESC,
    D3D10_MESSAGE_ID_CREATERENDERTARGETVIEW_INVALIDFORMAT,
    D3D10_MESSAGE_ID_CREATERENDERTARGETVIEW_INVALIDDIMENSIONS,
    D3D10_MESSAGE_ID_CREATERENDERTARGETVIEW_INVALIDRESOURCE,
    D3D10_MESSAGE_ID_CREATERENDERTARGETVIEW_TOOMANYOBJECTS,
    D3D10_MESSAGE_ID_CREATERENDERTARGETVIEW_INVALIDARG_RETURN,
    D3D10_MESSAGE_ID_CREATERENDERTARGETVIEW_OUTOFMEMORY_RETURN,
    D3D10_MESSAGE_ID_CREATEDEPTHSTENCILVIEW_UNRECOGNIZEDFORMAT,
    D3D10_MESSAGE_ID_CREATEDEPTHSTENCILVIEW_INVALIDDESC,
```

```
  D3D10_MESSAGE_ID_CREATEDEPTHSTENCILVIEW_INVALIDFORMAT,
  D3D10_MESSAGE_ID_CREATEDEPTHSTENCILVIEW_INVALIDDIMENSIONS,
  D3D10_MESSAGE_ID_CREATEDEPTHSTENCILVIEW_INVALIDRESOURCE,
  D3D10_MESSAGE_ID_CREATEDEPTHSTENCILVIEW_TOOMANYOBJECTS,
  D3D10_MESSAGE_ID_CREATEDEPTHSTENCILVIEW_INVALIDARG_RETURN,
  D3D10_MESSAGE_ID_CREATEDEPTHSTENCILVIEW_OUTOFMEMORY_RETURN,
  D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_OUTOFMEMORY,
  D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_TOOMANYELEMENTS,
  D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_INVALIDFORMAT,
  D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_INCOMPATIBLEFORMAT,
  D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_INVALIDSLOT,
  D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_INVALIDINPUTSLOTCLASS,
  D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_STEPRATESLOTCLASSMISMATCH,
  D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_INVALIDSLOTCLASSCHANGE,
  D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_INVALIDSTEPRATECHANGE,
  D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_INVALIDALIGNMENT,
  D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_DUPLICATESEMANTIC,
  D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_UNPARSEABLEINPUTSIGNATURE,
  D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_NULLSEMANTIC,
  D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_MISSINGELEMENT,
  D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_NULLDESC,
  D3D10_MESSAGE_ID_CREATEVERTEXSHADER_OUTOFMEMORY,
  D3D10_MESSAGE_ID_CREATEVERTEXSHADER_INVALIDSHADERBYTECODE,
  D3D10_MESSAGE_ID_CREATEVERTEXSHADER_INVALIDSHADERTYPE,
  D3D10_MESSAGE_ID_CREATEGEOMETRYSHADER_OUTOFMEMORY,
  D3D10_MESSAGE_ID_CREATEGEOMETRYSHADER_INVALIDSHADERBYTECODE,
  D3D10_MESSAGE_ID_CREATEGEOMETRYSHADER_INVALIDSHADERTYPE,
  D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_OUTOFMEMORY,

D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_INVALIDSHADERBYTECODE,
  D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_INVALIDSHADERTYPE,
  D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_INVALIDNUMENTRIES,

D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_OUTPUTSTREAMSTRIDEUNUS
ED,
  D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_UNEXPECTEDDECL,
  D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_EXPECTEDDECL,
  D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_OUTPUTSLOT0EXPECTED,
  D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_INVALIDOUTPUTSLOT,

D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_ONLYONEELEMENTPERSLOT,

D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_INVALIDCOMPONENTCOUNT,

D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_INVALIDSTARTCOMPONENTA
NDCOMPONENTCOUNT,

D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_INVALIDGAPDEFINITION,
  D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_REPEATEDOUTPUT,

D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_INVALIDOUTPUTSTREAMSTR
IDE,
  D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_MISSINGSEMANTIC,
  D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_MASKMISMATCH,
  D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_CANTHAVEONLYGAPS,
```

```
  D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_DECLTOOCOMPLEX,

D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_MISSINGOUTPUTSIGNATURE
,
  D3D10_MESSAGE_ID_CREATEPIXELSHADER_OUTOFMEMORY,
  D3D10_MESSAGE_ID_CREATEPIXELSHADER_INVALIDSHADERBYTECODE,
  D3D10_MESSAGE_ID_CREATEPIXELSHADER_INVALIDSHADERTYPE,
  D3D10_MESSAGE_ID_CREATERASTERIZERSTATE_INVALIDFILLMODE,
  D3D10_MESSAGE_ID_CREATERASTERIZERSTATE_INVALIDCULLMODE,
  D3D10_MESSAGE_ID_CREATERASTERIZERSTATE_INVALIDDEPTHBIASCLAMP,
  D3D10_MESSAGE_ID_CREATERASTERIZERSTATE_INVALIDSLOPESCALEDDEPTHBIAS,
  D3D10_MESSAGE_ID_CREATERASTERIZERSTATE_TOOMANYOBJECTS,
  D3D10_MESSAGE_ID_CREATERASTERIZERSTATE_NULLDESC,
  D3D10_MESSAGE_ID_CREATEDEPTHSTENCILSTATE_INVALIDDEPTHWRITEMASK,
  D3D10_MESSAGE_ID_CREATEDEPTHSTENCILSTATE_INVALIDDEPTHFUNC,
  D3D10_MESSAGE_ID_CREATEDEPTHSTENCILSTATE_INVALIDFRONTFACESTENCILFAILOP,
  D3D10_MESSAGE_ID_CREATEDEPTHSTENCILSTATE_INVALIDFRONTFACESTENCILZFAILOP,
  D3D10_MESSAGE_ID_CREATEDEPTHSTENCILSTATE_INVALIDFRONTFACESTENCILPASSOP,
  D3D10_MESSAGE_ID_CREATEDEPTHSTENCILSTATE_INVALIDFRONTFACESTENCILFUNC,
  D3D10_MESSAGE_ID_CREATEDEPTHSTENCILSTATE_INVALIDBACKFACESTENCILFAILOP,
  D3D10_MESSAGE_ID_CREATEDEPTHSTENCILSTATE_INVALIDBACKFACESTENCILZFAILOP,
  D3D10_MESSAGE_ID_CREATEDEPTHSTENCILSTATE_INVALIDBACKFACESTENCILPASSOP,
  D3D10_MESSAGE_ID_CREATEDEPTHSTENCILSTATE_INVALIDBACKFACESTENCILFUNC,
  D3D10_MESSAGE_ID_CREATEDEPTHSTENCILSTATE_TOOMANYOBJECTS,
  D3D10_MESSAGE_ID_CREATEDEPTHSTENCILSTATE_NULLDESC,
  D3D10_MESSAGE_ID_CREATEBLENDSTATE_INVALIDSRCBLEND,
  D3D10_MESSAGE_ID_CREATEBLENDSTATE_INVALIDDESTBLEND,
  D3D10_MESSAGE_ID_CREATEBLENDSTATE_INVALIDBLENDOP,
  D3D10_MESSAGE_ID_CREATEBLENDSTATE_INVALIDSRCBLENDALPHA,
  D3D10_MESSAGE_ID_CREATEBLENDSTATE_INVALIDDESTBLENDALPHA,
  D3D10_MESSAGE_ID_CREATEBLENDSTATE_INVALIDBLENDOPALPHA,
  D3D10_MESSAGE_ID_CREATEBLENDSTATE_INVALIDRENDERTARGETWRITEMASK,
  D3D10_MESSAGE_ID_CREATEBLENDSTATE_TOOMANYOBJECTS,
  D3D10_MESSAGE_ID_CREATEBLENDSTATE_NULLDESC,
  D3D10_MESSAGE_ID_CREATESAMPLERSTATE_INVALIDFILTER,
  D3D10_MESSAGE_ID_CREATESAMPLERSTATE_INVALIDADDRESSU,
  D3D10_MESSAGE_ID_CREATESAMPLERSTATE_INVALIDADDRESSV,
  D3D10_MESSAGE_ID_CREATESAMPLERSTATE_INVALIDADDRESSW,
  D3D10_MESSAGE_ID_CREATESAMPLERSTATE_INVALIDMIPLODBIAS,
  D3D10_MESSAGE_ID_CREATESAMPLERSTATE_INVALIDMAXANISOTROPY,
  D3D10_MESSAGE_ID_CREATESAMPLERSTATE_INVALIDCOMPARISONFUNC,
  D3D10_MESSAGE_ID_CREATESAMPLERSTATE_INVALIDMINLOD,
  D3D10_MESSAGE_ID_CREATESAMPLERSTATE_INVALIDMAXLOD,
  D3D10_MESSAGE_ID_CREATESAMPLERSTATE_TOOMANYOBJECTS,
  D3D10_MESSAGE_ID_CREATESAMPLERSTATE_NULLDESC,
  D3D10_MESSAGE_ID_CREATEQUERYORPREDICATE_INVALIDQUERY,
  D3D10_MESSAGE_ID_CREATEQUERYORPREDICATE_INVALIDMISCFLAGS,
  D3D10_MESSAGE_ID_CREATEQUERYORPREDICATE_UNEXPECTEDMISCFLAG,
  D3D10_MESSAGE_ID_CREATEQUERYORPREDICATE_NULLDESC,
  D3D10_MESSAGE_ID_DEVICE_IASETPRIMITIVETOPOLOGY_TOPOLOGY_UNRECOGNIZED,
  D3D10_MESSAGE_ID_DEVICE_IASETPRIMITIVETOPOLOGY_TOPOLOGY_UNDEFINED,
  D3D10_MESSAGE_ID_IASETVERTEXBUFFERS_INVALIDBUFFER,
  D3D10_MESSAGE_ID_DEVICE_IASETVERTEXBUFFERS_OFFSET_TOO_LARGE,
  D3D10_MESSAGE_ID_DEVICE_IASETVERTEXBUFFERS_BUFFERS_EMPTY,
  D3D10_MESSAGE_ID_IASETINDEXBUFFER_INVALIDBUFFER,
```

```
  D3D10_MESSAGE_ID_DEVICE_IASETINDEXBUFFER_FORMAT_INVALID,
  D3D10_MESSAGE_ID_DEVICE_IASETINDEXBUFFER_OFFSET_TOO_LARGE,
  D3D10_MESSAGE_ID_DEVICE_IASETINDEXBUFFER_OFFSET_UNALIGNED,
  D3D10_MESSAGE_ID_DEVICE_VSSETSHADERRESOURCES_VIEWS_EMPTY,
  D3D10_MESSAGE_ID_VSSETCONSTANTBUFFERS_INVALIDBUFFER,
  D3D10_MESSAGE_ID_DEVICE_VSSETCONSTANTBUFFERS_BUFFERS_EMPTY,
  D3D10_MESSAGE_ID_DEVICE_VSSETSAMPLERS_SAMPLERS_EMPTY,
  D3D10_MESSAGE_ID_DEVICE_GSSETSHADERRESOURCES_VIEWS_EMPTY,
  D3D10_MESSAGE_ID_GSSETCONSTANTBUFFERS_INVALIDBUFFER,
  D3D10_MESSAGE_ID_DEVICE_GSSETCONSTANTBUFFERS_BUFFERS_EMPTY,
  D3D10_MESSAGE_ID_DEVICE_GSSETSAMPLERS_SAMPLERS_EMPTY,
  D3D10_MESSAGE_ID_SOSETTARGETS_INVALIDBUFFER,
  D3D10_MESSAGE_ID_DEVICE_SOSETTARGETS_OFFSET_UNALIGNED,
  D3D10_MESSAGE_ID_DEVICE_PSSETSHADERRESOURCES_VIEWS_EMPTY,
  D3D10_MESSAGE_ID_PSSETCONSTANTBUFFERS_INVALIDBUFFER,
  D3D10_MESSAGE_ID_DEVICE_PSSETCONSTANTBUFFERS_BUFFERS_EMPTY,
  D3D10_MESSAGE_ID_DEVICE_PSSETSAMPLERS_SAMPLERS_EMPTY,
  D3D10_MESSAGE_ID_DEVICE_RSSETVIEWPORTS_INVALIDVIEWPORT,
  D3D10_MESSAGE_ID_DEVICE_RSSETSCISSORRECTS_INVALIDSCISSOR,
  D3D10_MESSAGE_ID_CLEARRENDERTARGETVIEW_DENORMFLUSH,
  D3D10_MESSAGE_ID_CLEARDEPTHSTENCILVIEW_DENORMFLUSH,
  D3D10_MESSAGE_ID_CLEARDEPTHSTENCILVIEW_INVALID,
  D3D10_MESSAGE_ID_DEVICE_IAGETVERTEXBUFFERS_BUFFERS_EMPTY,
  D3D10_MESSAGE_ID_DEVICE_VSGETSHADERRESOURCES_VIEWS_EMPTY,
  D3D10_MESSAGE_ID_DEVICE_VSGETCONSTANTBUFFERS_BUFFERS_EMPTY,
  D3D10_MESSAGE_ID_DEVICE_VSGETSAMPLERS_SAMPLERS_EMPTY,
  D3D10_MESSAGE_ID_DEVICE_GSGETSHADERRESOURCES_VIEWS_EMPTY,
  D3D10_MESSAGE_ID_DEVICE_GSGETCONSTANTBUFFERS_BUFFERS_EMPTY,
  D3D10_MESSAGE_ID_DEVICE_GSGETSAMPLERS_SAMPLERS_EMPTY,
  D3D10_MESSAGE_ID_DEVICE_SOGETTARGETS_BUFFERS_EMPTY,
  D3D10_MESSAGE_ID_DEVICE_PSGETSHADERRESOURCES_VIEWS_EMPTY,
  D3D10_MESSAGE_ID_DEVICE_PSGETCONSTANTBUFFERS_BUFFERS_EMPTY,
  D3D10_MESSAGE_ID_DEVICE_PSGETSAMPLERS_SAMPLERS_EMPTY,
  D3D10_MESSAGE_ID_DEVICE_RSGETVIEWPORTS_VIEWPORTS_EMPTY,
  D3D10_MESSAGE_ID_DEVICE_RSGETSCISSORRECTS_RECTS_EMPTY,
  D3D10_MESSAGE_ID_DEVICE_GENERATEMIPS_RESOURCE_INVALID,
  D3D10_MESSAGE_ID_COPYSUBRESOURCEREGION_INVALIDDESTINATIONSUBRESOURCE,
  D3D10_MESSAGE_ID_COPYSUBRESOURCEREGION_INVALIDSOURCESUBRESOURCE,
  D3D10_MESSAGE_ID_COPYSUBRESOURCEREGION_INVALIDSOURCEBOX,
  D3D10_MESSAGE_ID_COPYSUBRESOURCEREGION_INVALIDSOURCE,
  D3D10_MESSAGE_ID_COPYSUBRESOURCEREGION_INVALIDDESTINATIONSTATE,
  D3D10_MESSAGE_ID_COPYSUBRESOURCEREGION_INVALIDSOURCESTATE,
  D3D10_MESSAGE_ID_COPYRESOURCE_INVALIDSOURCE,
  D3D10_MESSAGE_ID_COPYRESOURCE_INVALIDDESTINATIONSTATE,
  D3D10_MESSAGE_ID_COPYRESOURCE_INVALIDSOURCESTATE,
  D3D10_MESSAGE_ID_UPDATESUBRESOURCE_INVALIDDESTINATIONSUBRESOURCE,
  D3D10_MESSAGE_ID_UPDATESUBRESOURCE_INVALIDDESTINATIONBOX,
  D3D10_MESSAGE_ID_UPDATESUBRESOURCE_INVALIDDESTINATIONSTATE,
  D3D10_MESSAGE_ID_DEVICE_RESOLVESUBRESOURCE_DESTINATION_INVALID,

D3D10_MESSAGE_ID_DEVICE_RESOLVESUBRESOURCE_DESTINATION_SUBRESOURCE_INVALID,
  D3D10_MESSAGE_ID_DEVICE_RESOLVESUBRESOURCE_SOURCE_INVALID,
  D3D10_MESSAGE_ID_DEVICE_RESOLVESUBRESOURCE_SOURCE_SUBRESOURCE_INVALID,
  D3D10_MESSAGE_ID_DEVICE_RESOLVESUBRESOURCE_FORMAT_INVALID,
  D3D10_MESSAGE_ID_BUFFER_MAP_INVALIDMAPTYPE,
```

```
    D3D10_MESSAGE_ID_BUFFER_MAP_INVALIDFLAGS,
    D3D10_MESSAGE_ID_BUFFER_MAP_ALREADYMAPPED,
    D3D10_MESSAGE_ID_BUFFER_MAP_DEVICEREMOVED_RETURN,
    D3D10_MESSAGE_ID_BUFFER_UNMAP_NOTMAPPED,
    D3D10_MESSAGE_ID_TEXTURE1D_MAP_INVALIDMAPTYPE,
    D3D10_MESSAGE_ID_TEXTURE1D_MAP_INVALIDSUBRESOURCE,
    D3D10_MESSAGE_ID_TEXTURE1D_MAP_INVALIDFLAGS,
    D3D10_MESSAGE_ID_TEXTURE1D_MAP_ALREADYMAPPED,
    D3D10_MESSAGE_ID_TEXTURE1D_MAP_DEVICEREMOVED_RETURN,
    D3D10_MESSAGE_ID_TEXTURE1D_UNMAP_INVALIDSUBRESOURCE,
    D3D10_MESSAGE_ID_TEXTURE1D_UNMAP_NOTMAPPED,
    D3D10_MESSAGE_ID_TEXTURE2D_MAP_INVALIDMAPTYPE,
    D3D10_MESSAGE_ID_TEXTURE2D_MAP_INVALIDSUBRESOURCE,
    D3D10_MESSAGE_ID_TEXTURE2D_MAP_INVALIDFLAGS,
    D3D10_MESSAGE_ID_TEXTURE2D_MAP_ALREADYMAPPED,
    D3D10_MESSAGE_ID_TEXTURE2D_MAP_DEVICEREMOVED_RETURN,
    D3D10_MESSAGE_ID_TEXTURE2D_UNMAP_INVALIDSUBRESOURCE,
    D3D10_MESSAGE_ID_TEXTURE2D_UNMAP_NOTMAPPED,
    D3D10_MESSAGE_ID_TEXTURE3D_MAP_INVALIDMAPTYPE,
    D3D10_MESSAGE_ID_TEXTURE3D_MAP_INVALIDSUBRESOURCE,
    D3D10_MESSAGE_ID_TEXTURE3D_MAP_INVALIDFLAGS,
    D3D10_MESSAGE_ID_TEXTURE3D_MAP_ALREADYMAPPED,
    D3D10_MESSAGE_ID_TEXTURE3D_MAP_DEVICEREMOVED_RETURN,
    D3D10_MESSAGE_ID_TEXTURE3D_UNMAP_INVALIDSUBRESOURCE,
    D3D10_MESSAGE_ID_TEXTURE3D_UNMAP_NOTMAPPED,
    D3D10_MESSAGE_ID_CHECKFORMATSUPPORT_FORMAT_DEPRECATED,
    D3D10_MESSAGE_ID_CHECKMULTISAMPLEQUALITYLEVELS_FORMAT_DEPRECATED,
    D3D10_MESSAGE_ID_SETEXCEPTIONMODE_UNRECOGNIZEDFLAGS,
    D3D10_MESSAGE_ID_SETEXCEPTIONMODE_INVALIDARG_RETURN,
    D3D10_MESSAGE_ID_SETEXCEPTIONMODE_DEVICEREMOVED_RETURN,
    D3D10_MESSAGE_ID_REF_SIMULATING_INFINITELY_FAST_HARDWARE,
    D3D10_MESSAGE_ID_REF_THREADING_MODE,
    D3D10_MESSAGE_ID_REF_UMDRIVER_EXCEPTION,
    D3D10_MESSAGE_ID_REF_KMDRIVER_EXCEPTION,
    D3D10_MESSAGE_ID_REF_HARDWARE_EXCEPTION,
    D3D10_MESSAGE_ID_REF_ACCESSING_INDEXABLE_TEMP_OUT_OF_RANGE,
    D3D10_MESSAGE_ID_REF_PROBLEM_PARSING_SHADER,
    D3D10_MESSAGE_ID_REF_OUT_OF_MEMORY,
    D3D10_MESSAGE_ID_REF_INFO,
    D3D10_MESSAGE_ID_DEVICE_DRAW_VERTEXPOS_OVERFLOW,
    D3D10_MESSAGE_ID_DEVICE_DRAWINDEXED_INDEXPOS_OVERFLOW,
    D3D10_MESSAGE_ID_DEVICE_DRAWINSTANCED_VERTEXPOS_OVERFLOW,
    D3D10_MESSAGE_ID_DEVICE_DRAWINSTANCED_INSTANCEPOS_OVERFLOW,
    D3D10_MESSAGE_ID_DEVICE_DRAWINDEXEDINSTANCED_INSTANCEPOS_OVERFLOW,
    D3D10_MESSAGE_ID_DEVICE_DRAWINDEXEDINSTANCED_INDEXPOS_OVERFLOW,
    D3D10_MESSAGE_ID_DEVICE_DRAW_VERTEX_SHADER_NOT_SET,
    D3D10_MESSAGE_ID_DEVICE_SHADER_LINKAGE_SEMANTICNAME_NOT_FOUND,
    D3D10_MESSAGE_ID_DEVICE_SHADER_LINKAGE_REGISTERINDEX,
    D3D10_MESSAGE_ID_DEVICE_SHADER_LINKAGE_COMPONENTTYPE,
    D3D10_MESSAGE_ID_DEVICE_SHADER_LINKAGE_REGISTERMASK,
    D3D10_MESSAGE_ID_DEVICE_SHADER_LINKAGE_SYSTEMVALUE,
    D3D10_MESSAGE_ID_DEVICE_SHADER_LINKAGE_NEVERWRITTEN_ALWAYSREADS,
    D3D10_MESSAGE_ID_DEVICE_DRAW_VERTEX_BUFFER_NOT_SET,
    D3D10_MESSAGE_ID_DEVICE_DRAW_INPUTLAYOUT_NOT_SET,
    D3D10_MESSAGE_ID_DEVICE_DRAW_CONSTANT_BUFFER_NOT_SET,
```

```
  D3D10_MESSAGE_ID_DEVICE_DRAW_CONSTANT_BUFFER_TOO_SMALL,
  D3D10_MESSAGE_ID_DEVICE_DRAW_SAMPLER_NOT_SET,
  D3D10_MESSAGE_ID_DEVICE_DRAW_SHADERRESOURCEVIEW_NOT_SET,
  D3D10_MESSAGE_ID_DEVICE_DRAW_VIEW_DIMENSION_MISMATCH,
  D3D10_MESSAGE_ID_DEVICE_DRAW_VERTEX_BUFFER_STRIDE_TOO_SMALL,
  D3D10_MESSAGE_ID_DEVICE_DRAW_VERTEX_BUFFER_TOO_SMALL,
  D3D10_MESSAGE_ID_DEVICE_DRAW_INDEX_BUFFER_NOT_SET,
  D3D10_MESSAGE_ID_DEVICE_DRAW_INDEX_BUFFER_FORMAT_INVALID,
  D3D10_MESSAGE_ID_DEVICE_DRAW_INDEX_BUFFER_TOO_SMALL,
  D3D10_MESSAGE_ID_DEVICE_DRAW_GS_INPUT_PRIMITIVE_MISMATCH,
  D3D10_MESSAGE_ID_DEVICE_DRAW_RESOURCE_RETURN_TYPE_MISMATCH,
  D3D10_MESSAGE_ID_DEVICE_DRAW_POSITION_NOT_PRESENT,
  D3D10_MESSAGE_ID_DEVICE_DRAW_OUTPUT_STREAM_NOT_SET,
  D3D10_MESSAGE_ID_DEVICE_DRAW_BOUND_RESOURCE_MAPPED,
  D3D10_MESSAGE_ID_DEVICE_DRAW_INVALID_PRIMITIVETOPOLOGY,
  D3D10_MESSAGE_ID_DEVICE_DRAW_VERTEX_OFFSET_UNALIGNED,
  D3D10_MESSAGE_ID_DEVICE_DRAW_VERTEX_STRIDE_UNALIGNED,
  D3D10_MESSAGE_ID_DEVICE_DRAW_INDEX_OFFSET_UNALIGNED,
  D3D10_MESSAGE_ID_DEVICE_DRAW_OUTPUT_STREAM_OFFSET_UNALIGNED,
  D3D10_MESSAGE_ID_DEVICE_DRAW_RESOURCE_FORMAT_LD_UNSUPPORTED,
  D3D10_MESSAGE_ID_DEVICE_DRAW_RESOURCE_FORMAT_SAMPLE_UNSUPPORTED,
  D3D10_MESSAGE_ID_DEVICE_DRAW_RESOURCE_FORMAT_SAMPLE_C_UNSUPPORTED,
  D3D10_MESSAGE_ID_DEVICE_DRAW_RESOURCE_MULTISAMPLE_UNSUPPORTED,
  D3D10_MESSAGE_ID_DEVICE_DRAW_SO_TARGETS_BOUND_WITHOUT_SOURCE,
  D3D10_MESSAGE_ID_DEVICE_DRAW_SO_STRIDE_LARGER_THAN_BUFFER,
  D3D10_MESSAGE_ID_DEVICE_DRAW_OM_RENDER_TARGET_DOES_NOT_SUPPORT_BLENDING,

D3D10_MESSAGE_ID_DEVICE_DRAW_OM_DUAL_SOURCE_BLENDING_CAN_ONLY_HAVE_RENDER_TA
RGET_0,
  D3D10_MESSAGE_ID_DEVICE_REMOVAL_PROCESS_AT_FAULT,
  D3D10_MESSAGE_ID_DEVICE_REMOVAL_PROCESS_POSSIBLY_AT_FAULT,
  D3D10_MESSAGE_ID_DEVICE_REMOVAL_PROCESS_NOT_AT_FAULT,
  D3D10_MESSAGE_ID_DEVICE_OPEN_SHARED_RESOURCE_INVALIDARG_RETURN,
  D3D10_MESSAGE_ID_DEVICE_OPEN_SHARED_RESOURCE_OUTOFMEMORY_RETURN,
  D3D10_MESSAGE_ID_DEVICE_OPEN_SHARED_RESOURCE_BADINTERFACE_RETURN,
  D3D10_MESSAGE_ID_DEVICE_DRAW_VIEWPORT_NOT_SET,
  D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_TRAILING_DIGIT_IN_SEMANTIC,

D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_TRAILING_DIGIT_IN_SEMA
NTIC,
  D3D10_MESSAGE_ID_DEVICE_RSSETVIEWPORTS_DENORMFLUSH,
  D3D10_MESSAGE_ID_OMSETRENDERTARGETS_INVALIDVIEW,
  D3D10_MESSAGE_ID_DEVICE_SETTEXTFILTERSIZE_INVALIDDIMENSIONS,
  D3D10_MESSAGE_ID_DEVICE_DRAW_SAMPLER_MISMATCH,
  D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_TYPE_MISMATCH,
  D3D10_MESSAGE_ID_BLENDSTATE_GETDESC_LEGACY,
  D3D10_MESSAGE_ID_SHADERRESOURCEVIEW_GETDESC_LEGACY,
  D3D10_MESSAGE_ID_CREATEQUERY_OUTOFMEMORY_RETURN,
  D3D10_MESSAGE_ID_CREATEPREDICATE_OUTOFMEMORY_RETURN,
  D3D10_MESSAGE_ID_CREATECOUNTER_OUTOFRANGE_COUNTER,
  D3D10_MESSAGE_ID_CREATECOUNTER_SIMULTANEOUS_ACTIVE_COUNTERS_EXHAUSTED,
  D3D10_MESSAGE_ID_CREATECOUNTER_UNSUPPORTED_WELLKNOWN_COUNTER,
  D3D10_MESSAGE_ID_CREATECOUNTER_OUTOFMEMORY_RETURN,
  D3D10_MESSAGE_ID_CREATECOUNTER_NONEXCLUSIVE_RETURN,
  D3D10_MESSAGE_ID_CREATECOUNTER_NULLDESC,
```

```
    D3D10_MESSAGE_ID_CHECKCOUNTER_OUTOFRANGE_COUNTER,
    D3D10_MESSAGE_ID_CHECKCOUNTER_UNSUPPORTED_WELLKNOWN_COUNTER,
    D3D10_MESSAGE_ID_SETPREDICATION_INVALID_PREDICATE_STATE,
    D3D10_MESSAGE_ID_QUERY_BEGIN_UNSUPPORTED,
    D3D10_MESSAGE_ID_PREDICATE_BEGIN_DURING_PREDICATION,
    D3D10_MESSAGE_ID_QUERY_BEGIN_DUPLICATE,
    D3D10_MESSAGE_ID_QUERY_BEGIN_ABANDONING_PREVIOUS_RESULTS,
    D3D10_MESSAGE_ID_PREDICATE_END_DURING_PREDICATION,
    D3D10_MESSAGE_ID_QUERY_END_ABANDONING_PREVIOUS_RESULTS,
    D3D10_MESSAGE_ID_QUERY_END_WITHOUT_BEGIN,
    D3D10_MESSAGE_ID_QUERY_GETDATA_INVALID_DATASIZE,
    D3D10_MESSAGE_ID_QUERY_GETDATA_INVALID_FLAGS,
    D3D10_MESSAGE_ID_QUERY_GETDATA_INVALID_CALL,
    D3D10_MESSAGE_ID_DEVICE_DRAW_PS_OUTPUT_TYPE_MISMATCH,
    D3D10_MESSAGE_ID_DEVICE_DRAW_RESOURCE_FORMAT_GATHER_UNSUPPORTED,
    D3D10_MESSAGE_ID_DEVICE_DRAW_INVALID_USE_OF_CENTER_MULTISAMPLE_PATTERN,
    D3D10_MESSAGE_ID_DEVICE_IASETVERTEXBUFFERS_STRIDE_TOO_LARGE,
    D3D10_MESSAGE_ID_DEVICE_IASETVERTEXBUFFERS_INVALIDRANGE,
    D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_EMPTY_LAYOUT,
    D3D10_MESSAGE_ID_DEVICE_DRAW_RESOURCE_SAMPLE_COUNT_MISMATCH,
    D3D10_MESSAGE_ID_LIVE_OBJECT_SUMMARY,
    D3D10_MESSAGE_ID_LIVE_BUFFER,
    D3D10_MESSAGE_ID_LIVE_TEXTURE1D,
    D3D10_MESSAGE_ID_LIVE_TEXTURE2D,
    D3D10_MESSAGE_ID_LIVE_TEXTURE3D,
    D3D10_MESSAGE_ID_LIVE_SHADERRESOURCEVIEW,
    D3D10_MESSAGE_ID_LIVE_RENDERTARGETVIEW,
    D3D10_MESSAGE_ID_LIVE_DEPTHSTENCILVIEW,
    D3D10_MESSAGE_ID_LIVE_VERTEXSHADER,
    D3D10_MESSAGE_ID_LIVE_GEOMETRYSHADER,
    D3D10_MESSAGE_ID_LIVE_PIXELSHADER,
    D3D10_MESSAGE_ID_LIVE_INPUTLAYOUT,
    D3D10_MESSAGE_ID_LIVE_SAMPLER,
    D3D10_MESSAGE_ID_LIVE_BLENDSTATE,
    D3D10_MESSAGE_ID_LIVE_DEPTHSTENCILSTATE,
    D3D10_MESSAGE_ID_LIVE_RASTERIZERSTATE,
    D3D10_MESSAGE_ID_LIVE_QUERY,
    D3D10_MESSAGE_ID_LIVE_PREDICATE,
    D3D10_MESSAGE_ID_LIVE_COUNTER,
    D3D10_MESSAGE_ID_LIVE_DEVICE,
    D3D10_MESSAGE_ID_LIVE_SWAPCHAIN,
    D3D10_MESSAGE_ID_D3D10_MESSAGES_END,
    D3D10_MESSAGE_ID_D3D10L9_MESSAGES_START = 0x100000,
    D3D10_MESSAGE_ID_CREATEDEPTHSTENCILSTATE_STENCIL_NO_TWO_SIDED,
    D3D10_MESSAGE_ID_CREATERASTERIZERSTATE_DepthBiasClamp_NOT_SUPPORTED,
    D3D10_MESSAGE_ID_CREATESAMPLERSTATE_NO_COMPARISON_SUPPORT,
    D3D10_MESSAGE_ID_CREATESAMPLERSTATE_EXCESSIVE_ANISOTROPY,
    D3D10_MESSAGE_ID_CREATESAMPLERSTATE_BORDER_OUT_OF_RANGE,
    D3D10_MESSAGE_ID_VSSETSAMPLERS_NOT_SUPPORTED,
    D3D10_MESSAGE_ID_VSSETSAMPLERS_TOO_MANY_SAMPLERS,
    D3D10_MESSAGE_ID_PSSETSAMPLERS_TOO_MANY_SAMPLERS,
    D3D10_MESSAGE_ID_CREATERESOURCE_NO_ARRAYS,
    D3D10_MESSAGE_ID_CREATERESOURCE_NO_VB_AND_IB_BIND,
    D3D10_MESSAGE_ID_CREATERESOURCE_NO_TEXTURE_1D,
    D3D10_MESSAGE_ID_CREATERESOURCE_DIMENSION_OUT_OF_RANGE,
```

```
  D3D10_MESSAGE_ID_CREATERESOURCE_NOT_BINDABLE_AS_SHADER_RESOURCE,
  D3D10_MESSAGE_ID_OMSETRENDERTARGETS_TOO_MANY_RENDER_TARGETS,
  D3D10_MESSAGE_ID_OMSETRENDERTARGETS_NO_DIFFERING_BIT_DEPTHS,
  D3D10_MESSAGE_ID_IASETVERTEXBUFFERS_BAD_BUFFER_INDEX,
  D3D10_MESSAGE_ID_DEVICE_RSSETVIEWPORTS_TOO_MANY_VIEWPORTS,
  D3D10_MESSAGE_ID_DEVICE_IASETPRIMITIVETOPOLOGY_ADJACENCY_UNSUPPORTED,
  D3D10_MESSAGE_ID_DEVICE_RSSETSCISSORRECTS_TOO_MANY_SCISSORS,
  D3D10_MESSAGE_ID_COPYRESOURCE_ONLY_TEXTURE_2D_WITHIN_GPU_MEMORY,
  D3D10_MESSAGE_ID_COPYRESOURCE_NO_TEXTURE_3D_READBACK,
  D3D10_MESSAGE_ID_COPYRESOURCE_NO_TEXTURE_ONLY_READBACK,
  D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_UNSUPPORTED_FORMAT,
  D3D10_MESSAGE_ID_CREATEBLENDSTATE_NO_ALPHA_TO_COVERAGE,
  D3D10_MESSAGE_ID_CREATERASTERIZERSTATE_DepthClipEnable_MUST_BE_TRUE,
  D3D10_MESSAGE_ID_DRAWINDEXED_STARTINDEXLOCATION_MUST_BE_POSITIVE,
  D3D10_MESSAGE_ID_CREATESHADERRESOURCEVIEW_MUST_USE_LOWEST_LOD,
  D3D10_MESSAGE_ID_CREATESAMPLERSTATE_MINLOD_MUST_NOT_BE_FRACTIONAL,
  D3D10_MESSAGE_ID_CREATESAMPLERSTATE_MAXLOD_MUST_BE_FLT_MAX,
  D3D10_MESSAGE_ID_CREATESHADERRESOURCEVIEW_FIRSTARRAYSLICE_MUST_BE_ZERO,
  D3D10_MESSAGE_ID_CREATESHADERRESOURCEVIEW_CUBES_MUST_HAVE_6_SIDES,
  D3D10_MESSAGE_ID_CREATERESOURCE_NOT_BINDABLE_AS_RENDER_TARGET,
  D3D10_MESSAGE_ID_CREATERESOURCE_NO_DWORD_INDEX_BUFFER,
  D3D10_MESSAGE_ID_CREATERESOURCE_MSAA_PRECLUDES_SHADER_RESOURCE,
  D3D10_MESSAGE_ID_CREATERESOURCE_PRESENTATION_PRECLUDES_SHADER_RESOURCE,
  D3D10_MESSAGE_ID_CREATEBLENDSTATE_NO_INDEPENDENT_BLEND_ENABLE,
  D3D10_MESSAGE_ID_CREATEBLENDSTATE_NO_INDEPENDENT_WRITE_MASKS,
  D3D10_MESSAGE_ID_CREATERESOURCE_NO_STREAM_OUT,
  D3D10_MESSAGE_ID_CREATERESOURCE_ONLY_VB_IB_FOR_BUFFERS,
  D3D10_MESSAGE_ID_CREATERESOURCE_NO_AUTOGEN_FOR_VOLUMES,
  D3D10_MESSAGE_ID_CREATERESOURCE_DXGI_FORMAT_R8G8B8A8_CANNOT_BE_SHARED,
  D3D10_MESSAGE_ID_VSSHADERRESOURCES_NOT_SUPPORTED,
  D3D10_MESSAGE_ID_GEOMETRY_SHADER_NOT_SUPPORTED,
  D3D10_MESSAGE_ID_STREAM_OUT_NOT_SUPPORTED,
  D3D10_MESSAGE_ID_TEXT_FILTER_NOT_SUPPORTED,
  D3D10_MESSAGE_ID_CREATEBLENDSTATE_NO_SEPARATE_ALPHA_BLEND,
  D3D10_MESSAGE_ID_CREATEBLENDSTATE_NO_MRT_BLEND,
  D3D10_MESSAGE_ID_CREATEBLENDSTATE_OPERATION_NOT_SUPPORTED,
  D3D10_MESSAGE_ID_CREATESAMPLERSTATE_NO_MIRRORONCE,
  D3D10_MESSAGE_ID_DRAWINSTANCED_NOT_SUPPORTED,
  D3D10_MESSAGE_ID_DRAWINDEXEDINSTANCED_NOT_SUPPORTED_BELOW_9_3,
  D3D10_MESSAGE_ID_DRAWINDEXED_POINTLIST_UNSUPPORTED,
  D3D10_MESSAGE_ID_SETBLENDSTATE_SAMPLE_MASK_CANNOT_BE_ZERO,

D3D10_MESSAGE_ID_CREATERESOURCE_DIMENSION_EXCEEDS_FEATURE_LEVEL_DEFINITION,

D3D10_MESSAGE_ID_CREATERESOURCE_ONLY_SINGLE_MIP_LEVEL_DEPTH_STENCIL_SUPPORTE
D,
  D3D10_MESSAGE_ID_DEVICE_RSSETSCISSORRECTS_NEGATIVESCISSOR,
  D3D10_MESSAGE_ID_SLOT_ZERO_MUST_BE_D3D10_INPUT_PER_VERTEX_DATA,
  D3D10_MESSAGE_ID_CREATERESOURCE_NON_POW_2_MIPMAP,
  D3D10_MESSAGE_ID_CREATESAMPLERSTATE_BORDER_NOT_SUPPORTED,
  D3D10_MESSAGE_ID_OMSETRENDERTARGETS_NO_SRGB_MRT,
  D3D10_MESSAGE_ID_COPYRESOURCE_NO_3D_MISMATCHED_UPDATES,
  D3D10_MESSAGE_ID_D3D10L9_MESSAGES_END
} ;
```

Constants

`D3D10_MESSAGE_ID_UNKNOWN` Value: *0*
`D3D10_MESSAGE_ID_DEVICE_IASETVERTEXBUFFERS_HAZARD`
`D3D10_MESSAGE_ID_DEVICE_IASETINDEXBUFFER_HAZARD`
`D3D10_MESSAGE_ID_DEVICE_VSSETSHADERRESOURCES_HAZARD`
`D3D10_MESSAGE_ID_DEVICE_VSSETCONSTANTBUFFERS_HAZARD`
`D3D10_MESSAGE_ID_DEVICE_GSSETSHADERRESOURCES_HAZARD`
`D3D10_MESSAGE_ID_DEVICE_GSSETCONSTANTBUFFERS_HAZARD`
`D3D10_MESSAGE_ID_DEVICE_PSSETSHADERRESOURCES_HAZARD`
`D3D10_MESSAGE_ID_DEVICE_PSSETCONSTANTBUFFERS_HAZARD`
`D3D10_MESSAGE_ID_DEVICE_OMSETRENDERTARGETS_HAZARD`
`D3D10_MESSAGE_ID_DEVICE_SOSETTARGETS_HAZARD`
`D3D10_MESSAGE_ID_STRING_FROM_APPLICATION`
`D3D10_MESSAGE_ID_CORRUPTED_THIS`
`D3D10_MESSAGE_ID_CORRUPTED_PARAMETER1`
`D3D10_MESSAGE_ID_CORRUPTED_PARAMETER2`
`D3D10_MESSAGE_ID_CORRUPTED_PARAMETER3`
`D3D10_MESSAGE_ID_CORRUPTED_PARAMETER4`
`D3D10_MESSAGE_ID_CORRUPTED_PARAMETER5`
`D3D10_MESSAGE_ID_CORRUPTED_PARAMETER6`
`D3D10_MESSAGE_ID_CORRUPTED_PARAMETER7`
`D3D10_MESSAGE_ID_CORRUPTED_PARAMETER8`
`D3D10_MESSAGE_ID_CORRUPTED_PARAMETER9`
`D3D10_MESSAGE_ID_CORRUPTED_PARAMETER10`
`D3D10_MESSAGE_ID_CORRUPTED_PARAMETER11`

D3D10_MESSAGE_ID_CORRUPTED_PARAMETER12
D3D10_MESSAGE_ID_CORRUPTED_PARAMETER13
D3D10_MESSAGE_ID_CORRUPTED_PARAMETER14
D3D10_MESSAGE_ID_CORRUPTED_PARAMETER15
D3D10_MESSAGE_ID_CORRUPTED_MULTITHREADING
D3D10_MESSAGE_ID_MESSAGE_REPORTING_OUTOFMEMORY
D3D10_MESSAGE_ID_IASETINPUTLAYOUT_UNBINDDELETINGOBJECT
D3D10_MESSAGE_ID_IASETVERTEXBUFFERS_UNBINDDELETINGOBJECT
D3D10_MESSAGE_ID_IASETINDEXBUFFER_UNBINDDELETINGOBJECT
D3D10_MESSAGE_ID_VSSETSHADER_UNBINDDELETINGOBJECT
D3D10_MESSAGE_ID_VSSETSHADERRESOURCES_UNBINDDELETINGOBJECT
D3D10_MESSAGE_ID_VSSETCONSTANTBUFFERS_UNBINDDELETINGOBJECT
D3D10_MESSAGE_ID_VSSETSAMPLERS_UNBINDDELETINGOBJECT
D3D10_MESSAGE_ID_GSSETSHADER_UNBINDDELETINGOBJECT
D3D10_MESSAGE_ID_GSSETSHADERRESOURCES_UNBINDDELETINGOBJECT
D3D10_MESSAGE_ID_GSSETCONSTANTBUFFERS_UNBINDDELETINGOBJECT
D3D10_MESSAGE_ID_GSSETSAMPLERS_UNBINDDELETINGOBJECT
D3D10_MESSAGE_ID_SOSETTARGETS_UNBINDDELETINGOBJECT
D3D10_MESSAGE_ID_PSSETSHADER_UNBINDDELETINGOBJECT
D3D10_MESSAGE_ID_PSSETSHADERRESOURCES_UNBINDDELETINGOBJECT
D3D10_MESSAGE_ID_PSSETCONSTANTBUFFERS_UNBINDDELETINGOBJECT
D3D10_MESSAGE_ID_PSSETSAMPLERS_UNBINDDELETINGOBJECT
D3D10_MESSAGE_ID_RSSETSTATE_UNBINDDELETINGOBJECT
D3D10_MESSAGE_ID_OMSETBLENDSTATE_UNBINDDELETINGOBJECT
D3D10_MESSAGE_ID_OMSETDEPTHSTENCILSTATE_UNBINDDELETINGOBJECT

D3D10_MESSAGE_ID_OMSETRENDERTARGETS_UNBINDDELETINGOBJECT
D3D10_MESSAGE_ID_SETPREDICATION_UNBINDDELETINGOBJECT
D3D10_MESSAGE_ID_GETPRIVATEDATA_MOREDATA
D3D10_MESSAGE_ID_SETPRIVATEDATA_INVALIDFREEDATA
D3D10_MESSAGE_ID_SETPRIVATEDATA_INVALIDIUNKNOWN
D3D10_MESSAGE_ID_SETPRIVATEDATA_INVALIDFLAGS
D3D10_MESSAGE_ID_SETPRIVATEDATA_CHANGINGPARAMS
D3D10_MESSAGE_ID_SETPRIVATEDATA_OUTOFMEMORY
D3D10_MESSAGE_ID_CREATEBUFFER_UNRECOGNIZEDFORMAT
D3D10_MESSAGE_ID_CREATEBUFFER_INVALIDSAMPLES
D3D10_MESSAGE_ID_CREATEBUFFER_UNRECOGNIZEDUSAGE
D3D10_MESSAGE_ID_CREATEBUFFER_UNRECOGNIZEDBINDFLAGS
D3D10_MESSAGE_ID_CREATEBUFFER_UNRECOGNIZEDCPUACCESSFLAGS
D3D10_MESSAGE_ID_CREATEBUFFER_UNRECOGNIZEDMISCFLAGS
D3D10_MESSAGE_ID_CREATEBUFFER_INVALIDCPUACCESSFLAGS
D3D10_MESSAGE_ID_CREATEBUFFER_INVALIDBINDFLAGS
D3D10_MESSAGE_ID_CREATEBUFFER_INVALIDINITIALDATA
D3D10_MESSAGE_ID_CREATEBUFFER_INVALIDDIMENSIONS
D3D10_MESSAGE_ID_CREATEBUFFER_INVALIDMIPLEVELS
D3D10_MESSAGE_ID_CREATEBUFFER_INVALIDMISCFLAGS
D3D10_MESSAGE_ID_CREATEBUFFER_INVALIDARG_RETURN
D3D10_MESSAGE_ID_CREATEBUFFER_OUTOFMEMORY_RETURN
D3D10_MESSAGE_ID_CREATEBUFFER_NULLDESC
D3D10_MESSAGE_ID_CREATEBUFFER_INVALIDCONSTANTBUFFERBINDINGS
D3D10_MESSAGE_ID_CREATEBUFFER_LARGEALLOCATION

D3D10_MESSAGE_ID_CREATETEXTURE1D_UNRECOGNIZEDFORMAT
D3D10_MESSAGE_ID_CREATETEXTURE1D_UNSUPPORTEDFORMAT
D3D10_MESSAGE_ID_CREATETEXTURE1D_INVALIDSAMPLES
D3D10_MESSAGE_ID_CREATETEXTURE1D_UNRECOGNIZEDUSAGE
D3D10_MESSAGE_ID_CREATETEXTURE1D_UNRECOGNIZEDBINDFLAGS
D3D10_MESSAGE_ID_CREATETEXTURE1D_UNRECOGNIZEDCPUACCESSFLAGS
D3D10_MESSAGE_ID_CREATETEXTURE1D_UNRECOGNIZEDMISCFLAGS
D3D10_MESSAGE_ID_CREATETEXTURE1D_INVALIDCPUACCESSFLAGS
D3D10_MESSAGE_ID_CREATETEXTURE1D_INVALIDBINDFLAGS
D3D10_MESSAGE_ID_CREATETEXTURE1D_INVALIDINITIALDATA
D3D10_MESSAGE_ID_CREATETEXTURE1D_INVALIDDIMENSIONS
D3D10_MESSAGE_ID_CREATETEXTURE1D_INVALIDMIPLEVELS
D3D10_MESSAGE_ID_CREATETEXTURE1D_INVALIDMISCFLAGS
D3D10_MESSAGE_ID_CREATETEXTURE1D_INVALIDARG_RETURN
D3D10_MESSAGE_ID_CREATETEXTURE1D_OUTOFMEMORY_RETURN
D3D10_MESSAGE_ID_CREATETEXTURE1D_NULLDESC
D3D10_MESSAGE_ID_CREATETEXTURE1D_LARGEALLOCATION
D3D10_MESSAGE_ID_CREATETEXTURE2D_UNRECOGNIZEDFORMAT
D3D10_MESSAGE_ID_CREATETEXTURE2D_UNSUPPORTEDFORMAT
D3D10_MESSAGE_ID_CREATETEXTURE2D_INVALIDSAMPLES
D3D10_MESSAGE_ID_CREATETEXTURE2D_UNRECOGNIZEDUSAGE
D3D10_MESSAGE_ID_CREATETEXTURE2D_UNRECOGNIZEDBINDFLAGS
D3D10_MESSAGE_ID_CREATETEXTURE2D_UNRECOGNIZEDCPUACCESSFLAGS
D3D10_MESSAGE_ID_CREATETEXTURE2D_UNRECOGNIZEDMISCFLAGS
D3D10_MESSAGE_ID_CREATETEXTURE2D_INVALIDCPUACCESSFLAGS

D3D10_MESSAGE_ID_CREATETEXTURE2D_INVALIDBINDFLAGS
D3D10_MESSAGE_ID_CREATETEXTURE2D_INVALIDINITIALDATA
D3D10_MESSAGE_ID_CREATETEXTURE2D_INVALIDDIMENSIONS
D3D10_MESSAGE_ID_CREATETEXTURE2D_INVALIDMIPLEVELS
D3D10_MESSAGE_ID_CREATETEXTURE2D_INVALIDMISCFLAGS
D3D10_MESSAGE_ID_CREATETEXTURE2D_INVALIDARG_RETURN
D3D10_MESSAGE_ID_CREATETEXTURE2D_OUTOFMEMORY_RETURN
D3D10_MESSAGE_ID_CREATETEXTURE2D_NULLDESC
D3D10_MESSAGE_ID_CREATETEXTURE2D_LARGEALLOCATION
D3D10_MESSAGE_ID_CREATETEXTURE3D_UNRECOGNIZEDFORMAT
D3D10_MESSAGE_ID_CREATETEXTURE3D_UNSUPPORTEDFORMAT
D3D10_MESSAGE_ID_CREATETEXTURE3D_INVALIDSAMPLES
D3D10_MESSAGE_ID_CREATETEXTURE3D_UNRECOGNIZEDUSAGE
D3D10_MESSAGE_ID_CREATETEXTURE3D_UNRECOGNIZEDBINDFLAGS
D3D10_MESSAGE_ID_CREATETEXTURE3D_UNRECOGNIZEDCPUACCESSFLAGS
D3D10_MESSAGE_ID_CREATETEXTURE3D_UNRECOGNIZEDMISCFLAGS
D3D10_MESSAGE_ID_CREATETEXTURE3D_INVALIDCPUACCESSFLAGS
D3D10_MESSAGE_ID_CREATETEXTURE3D_INVALIDBINDFLAGS
D3D10_MESSAGE_ID_CREATETEXTURE3D_INVALIDINITIALDATA
D3D10_MESSAGE_ID_CREATETEXTURE3D_INVALIDDIMENSIONS
D3D10_MESSAGE_ID_CREATETEXTURE3D_INVALIDMIPLEVELS
D3D10_MESSAGE_ID_CREATETEXTURE3D_INVALIDMISCFLAGS
D3D10_MESSAGE_ID_CREATETEXTURE3D_INVALIDARG_RETURN
D3D10_MESSAGE_ID_CREATETEXTURE3D_OUTOFMEMORY_RETURN
D3D10_MESSAGE_ID_CREATETEXTURE3D_NULLDESC

D3D10_MESSAGE_ID_CREATETEXTURE3D_LARGEALLOCATION
D3D10_MESSAGE_ID_CREATESHADERRESOURCEVIEW_UNRECOGNIZEDFORMAT
D3D10_MESSAGE_ID_CREATESHADERRESOURCEVIEW_INVALIDDESC
D3D10_MESSAGE_ID_CREATESHADERRESOURCEVIEW_INVALIDFORMAT
D3D10_MESSAGE_ID_CREATESHADERRESOURCEVIEW_INVALIDDIMENSIONS
D3D10_MESSAGE_ID_CREATESHADERRESOURCEVIEW_INVALIDRESOURCE
D3D10_MESSAGE_ID_CREATESHADERRESOURCEVIEW_TOOMANYOBJECTS
D3D10_MESSAGE_ID_CREATESHADERRESOURCEVIEW_INVALIDARG_RETURN
D3D10_MESSAGE_ID_CREATESHADERRESOURCEVIEW_OUTOFMEMORY_RETURN
D3D10_MESSAGE_ID_CREATERENDERTARGETVIEW_UNRECOGNIZEDFORMAT
D3D10_MESSAGE_ID_CREATERENDERTARGETVIEW_UNSUPPORTEDFORMAT
D3D10_MESSAGE_ID_CREATERENDERTARGETVIEW_INVALIDDESC
D3D10_MESSAGE_ID_CREATERENDERTARGETVIEW_INVALIDFORMAT
D3D10_MESSAGE_ID_CREATERENDERTARGETVIEW_INVALIDDIMENSIONS
D3D10_MESSAGE_ID_CREATERENDERTARGETVIEW_INVALIDRESOURCE
D3D10_MESSAGE_ID_CREATERENDERTARGETVIEW_TOOMANYOBJECTS
D3D10_MESSAGE_ID_CREATERENDERTARGETVIEW_INVALIDARG_RETURN
D3D10_MESSAGE_ID_CREATERENDERTARGETVIEW_OUTOFMEMORY_RETURN
D3D10_MESSAGE_ID_CREATEDEPTHSTENCILVIEW_UNRECOGNIZEDFORMAT
D3D10_MESSAGE_ID_CREATEDEPTHSTENCILVIEW_INVALIDDESC
D3D10_MESSAGE_ID_CREATEDEPTHSTENCILVIEW_INVALIDFORMAT
D3D10_MESSAGE_ID_CREATEDEPTHSTENCILVIEW_INVALIDDIMENSIONS
D3D10_MESSAGE_ID_CREATEDEPTHSTENCILVIEW_INVALIDRESOURCE
D3D10_MESSAGE_ID_CREATEDEPTHSTENCILVIEW_TOOMANYOBJECTS
D3D10_MESSAGE_ID_CREATEDEPTHSTENCILVIEW_INVALIDARG_RETURN

D3D10_MESSAGE_ID_CREATEDEPTHSTENCILVIEW_OUTOFMEMORY_RETURN
D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_OUTOFMEMORY
D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_TOOMANYELEMENTS
D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_INVALIDFORMAT
D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_INCOMPATIBLEFORMAT
D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_INVALIDSLOT
D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_INVALIDINPUTSLOTCLASS
D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_STEPRATESLOTCLASSMISMATCH
D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_INVALIDSLOTCLASSCHANGE
D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_INVALIDSTEPRATECHANGE
D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_INVALIDALIGNMENT
D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_DUPLICATESEMANTIC
D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_UNPARSEABLEINPUTSIGNATURE
D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_NULLSEMANTIC
D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_MISSINGELEMENT
D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_NULLDESC
D3D10_MESSAGE_ID_CREATEVERTEXSHADER_OUTOFMEMORY
D3D10_MESSAGE_ID_CREATEVERTEXSHADER_INVALIDSHADERBYTECODE
D3D10_MESSAGE_ID_CREATEVERTEXSHADER_INVALIDSHADERTYPE
D3D10_MESSAGE_ID_CREATEGEOMETRYSHADER_OUTOFMEMORY
D3D10_MESSAGE_ID_CREATEGEOMETRYSHADER_INVALIDSHADERBYTECODE
D3D10_MESSAGE_ID_CREATEGEOMETRYSHADER_INVALIDSHADERTYPE
D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_OUTOFMEMORY
D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_INVALIDSHADERBYTECODE
D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_INVALIDSHADERTYPE

D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_INVALIDNUMENTRIES
D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_OUTPUTSTREAMSTRIDEUNUSED
D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_UNEXPECTEDDECL
D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_EXPECTEDDECL
D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_OUTPUTSLOT0EXPECTED
D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_INVALIDOUTPUTSLOT
D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_ONLYONEELEMENTPERSLOT
D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_INVALIDCOMPONENTCOUNT
D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_INVALIDSTARTCOMPONENTANDCOMPONENTCOUNT
D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_INVALIDGAPDEFINITION
D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_REPEATEDOUTPUT
D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_INVALIDOUTPUTSTREAMSTRIDE
D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_MISSINGSEMANTIC
D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_MASKMISMATCH
D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_CANTHAVEONLYGAPS
D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_DECLTOOCOMPLEX
D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_MISSINGOUTPUTSIGNATURE
D3D10_MESSAGE_ID_CREATEPIXELSHADER_OUTOFMEMORY
D3D10_MESSAGE_ID_CREATEPIXELSHADER_INVALIDSHADERBYTECODE
D3D10_MESSAGE_ID_CREATEPIXELSHADER_INVALIDSHADERTYPE
D3D10_MESSAGE_ID_CREATERASTERIZERSTATE_INVALIDFILLMODE
D3D10_MESSAGE_ID_CREATERASTERIZERSTATE_INVALIDCULLMODE
D3D10_MESSAGE_ID_CREATERASTERIZERSTATE_INVALIDDEPTHBIASCLAMP
D3D10_MESSAGE_ID_CREATERASTERIZERSTATE_INVALIDSLOPESCALEDDEPTHBIAS
D3D10_MESSAGE_ID_CREATERASTERIZERSTATE_TOOMANYOBJECTS

D3D10_MESSAGE_ID_CREATERASTERIZERSTATE_NULLDESC
D3D10_MESSAGE_ID_CREATEDEPTHSTENCILSTATE_INVALIDDEPTHWRITEMASK
D3D10_MESSAGE_ID_CREATEDEPTHSTENCILSTATE_INVALIDDEPTHFUNC
D3D10_MESSAGE_ID_CREATEDEPTHSTENCILSTATE_INVALIDFRONTFACESTENCILFAILOP
D3D10_MESSAGE_ID_CREATEDEPTHSTENCILSTATE_INVALIDFRONTFACESTENCILZFAILOP
D3D10_MESSAGE_ID_CREATEDEPTHSTENCILSTATE_INVALIDFRONTFACESTENCILPASSOP
D3D10_MESSAGE_ID_CREATEDEPTHSTENCILSTATE_INVALIDFRONTFACESTENCILFUNC
D3D10_MESSAGE_ID_CREATEDEPTHSTENCILSTATE_INVALIDBACKFACESTENCILFAILOP
D3D10_MESSAGE_ID_CREATEDEPTHSTENCILSTATE_INVALIDBACKFACESTENCILZFAILOP
D3D10_MESSAGE_ID_CREATEDEPTHSTENCILSTATE_INVALIDBACKFACESTENCILPASSOP
D3D10_MESSAGE_ID_CREATEDEPTHSTENCILSTATE_INVALIDBACKFACESTENCILFUNC
D3D10_MESSAGE_ID_CREATEDEPTHSTENCILSTATE_TOOMANYOBJECTS
D3D10_MESSAGE_ID_CREATEDEPTHSTENCILSTATE_NULLDESC
D3D10_MESSAGE_ID_CREATEBLENDSTATE_INVALIDSRCBLEND
D3D10_MESSAGE_ID_CREATEBLENDSTATE_INVALIDDESTBLEND
D3D10_MESSAGE_ID_CREATEBLENDSTATE_INVALIDBLENDOP
D3D10_MESSAGE_ID_CREATEBLENDSTATE_INVALIDSRCBLENDALPHA
D3D10_MESSAGE_ID_CREATEBLENDSTATE_INVALIDDESTBLENDALPHA
D3D10_MESSAGE_ID_CREATEBLENDSTATE_INVALIDBLENDOPALPHA
D3D10_MESSAGE_ID_CREATEBLENDSTATE_INVALIDRENDERTARGETWRITEMASK
D3D10_MESSAGE_ID_CREATEBLENDSTATE_TOOMANYOBJECTS
D3D10_MESSAGE_ID_CREATEBLENDSTATE_NULLDESC
D3D10_MESSAGE_ID_CREATESAMPLERSTATE_INVALIDFILTER
D3D10_MESSAGE_ID_CREATESAMPLERSTATE_INVALIDADDRESSU
D3D10_MESSAGE_ID_CREATESAMPLERSTATE_INVALIDADDRESSV

D3D10_MESSAGE_ID_CREATESAMPLERSTATE_INVALIDADDRESSW
D3D10_MESSAGE_ID_CREATESAMPLERSTATE_INVALIDMIPLODBIAS
D3D10_MESSAGE_ID_CREATESAMPLERSTATE_INVALIDMAXANISOTROPY
D3D10_MESSAGE_ID_CREATESAMPLERSTATE_INVALIDCOMPARISONFUNC
D3D10_MESSAGE_ID_CREATESAMPLERSTATE_INVALIDMINLOD
D3D10_MESSAGE_ID_CREATESAMPLERSTATE_INVALIDMAXLOD
D3D10_MESSAGE_ID_CREATESAMPLERSTATE_TOOMANYOBJECTS
D3D10_MESSAGE_ID_CREATESAMPLERSTATE_NULLDESC
D3D10_MESSAGE_ID_CREATEQUERYORPREDICATE_INVALIDQUERY
D3D10_MESSAGE_ID_CREATEQUERYORPREDICATE_INVALIDMISCFLAGS
D3D10_MESSAGE_ID_CREATEQUERYORPREDICATE_UNEXPECTEDMISCFLAG
D3D10_MESSAGE_ID_CREATEQUERYORPREDICATE_NULLDESC
D3D10_MESSAGE_ID_DEVICE_IASETPRIMITIVETOPOLOGY_TOPOLOGY_UNRECOGNIZED
D3D10_MESSAGE_ID_DEVICE_IASETPRIMITIVETOPOLOGY_TOPOLOGY_UNDEFINED
D3D10_MESSAGE_ID_IASETVERTEXBUFFERS_INVALIDBUFFER
D3D10_MESSAGE_ID_DEVICE_IASETVERTEXBUFFERS_OFFSET_TOO_LARGE
D3D10_MESSAGE_ID_DEVICE_IASETVERTEXBUFFERS_BUFFERS_EMPTY
D3D10_MESSAGE_ID_IASETINDEXBUFFER_INVALIDBUFFER
D3D10_MESSAGE_ID_DEVICE_IASETINDEXBUFFER_FORMAT_INVALID
D3D10_MESSAGE_ID_DEVICE_IASETINDEXBUFFER_OFFSET_TOO_LARGE
D3D10_MESSAGE_ID_DEVICE_IASETINDEXBUFFER_OFFSET_UNALIGNED
D3D10_MESSAGE_ID_DEVICE_VSSETSHADERRESOURCES_VIEWS_EMPTY
D3D10_MESSAGE_ID_VSSETCONSTANTBUFFERS_INVALIDBUFFER
D3D10_MESSAGE_ID_DEVICE_VSSETCONSTANTBUFFERS_BUFFERS_EMPTY
D3D10_MESSAGE_ID_DEVICE_VSSETSAMPLERS_SAMPLERS_EMPTY

D3D10_MESSAGE_ID_DEVICE_GSSETSHADERRESOURCES_VIEWS_EMPTY
D3D10_MESSAGE_ID_GSSETCONSTANTBUFFERS_INVALIDBUFFER
D3D10_MESSAGE_ID_DEVICE_GSSETCONSTANTBUFFERS_BUFFERS_EMPTY
D3D10_MESSAGE_ID_DEVICE_GSSETSAMPLERS_SAMPLERS_EMPTY
D3D10_MESSAGE_ID_SOSETTARGETS_INVALIDBUFFER
D3D10_MESSAGE_ID_DEVICE_SOSETTARGETS_OFFSET_UNALIGNED
D3D10_MESSAGE_ID_DEVICE_PSSETSHADERRESOURCES_VIEWS_EMPTY
D3D10_MESSAGE_ID_PSSETCONSTANTBUFFERS_INVALIDBUFFER
D3D10_MESSAGE_ID_DEVICE_PSSETCONSTANTBUFFERS_BUFFERS_EMPTY
D3D10_MESSAGE_ID_DEVICE_PSSETSAMPLERS_SAMPLERS_EMPTY
D3D10_MESSAGE_ID_DEVICE_RSSETVIEWPORTS_INVALIDVIEWPORT
D3D10_MESSAGE_ID_DEVICE_RSSETSCISSORRECTS_INVALIDSCISSOR
D3D10_MESSAGE_ID_CLEARRENDERTARGETVIEW_DENORMFLUSH
D3D10_MESSAGE_ID_CLEARDEPTHSTENCILVIEW_DENORMFLUSH
D3D10_MESSAGE_ID_CLEARDEPTHSTENCILVIEW_INVALID
D3D10_MESSAGE_ID_DEVICE_IAGETVERTEXBUFFERS_BUFFERS_EMPTY
D3D10_MESSAGE_ID_DEVICE_VSGETSHADERRESOURCES_VIEWS_EMPTY
D3D10_MESSAGE_ID_DEVICE_VSGETCONSTANTBUFFERS_BUFFERS_EMPTY
D3D10_MESSAGE_ID_DEVICE_VSGETSAMPLERS_SAMPLERS_EMPTY
D3D10_MESSAGE_ID_DEVICE_GSGETSHADERRESOURCES_VIEWS_EMPTY
D3D10_MESSAGE_ID_DEVICE_GSGETCONSTANTBUFFERS_BUFFERS_EMPTY
D3D10_MESSAGE_ID_DEVICE_GSGETSAMPLERS_SAMPLERS_EMPTY
D3D10_MESSAGE_ID_DEVICE_SOGETTARGETS_BUFFERS_EMPTY
D3D10_MESSAGE_ID_DEVICE_PSGETSHADERRESOURCES_VIEWS_EMPTY
D3D10_MESSAGE_ID_DEVICE_PSGETCONSTANTBUFFERS_BUFFERS_EMPTY

D3D10_MESSAGE_ID_DEVICE_PSGETSAMPLERS_SAMPLERS_EMPTY
D3D10_MESSAGE_ID_DEVICE_RSGETVIEWPORTS_VIEWPORTS_EMPTY
D3D10_MESSAGE_ID_DEVICE_RSGETSCISSORRECTS_RECTS_EMPTY
D3D10_MESSAGE_ID_DEVICE_GENERATEMIPS_RESOURCE_INVALID
D3D10_MESSAGE_ID_COPYSUBRESOURCEREGION_INVALIDDESTINATIONSUBRESOURCE
D3D10_MESSAGE_ID_COPYSUBRESOURCEREGION_INVALIDSOURCESUBRESOURCE
D3D10_MESSAGE_ID_COPYSUBRESOURCEREGION_INVALIDSOURCEBOX
D3D10_MESSAGE_ID_COPYSUBRESOURCEREGION_INVALIDSOURCE
D3D10_MESSAGE_ID_COPYSUBRESOURCEREGION_INVALIDDESTINATIONSTATE
D3D10_MESSAGE_ID_COPYSUBRESOURCEREGION_INVALIDSOURCESTATE
D3D10_MESSAGE_ID_COPYRESOURCE_INVALIDSOURCE
D3D10_MESSAGE_ID_COPYRESOURCE_INVALIDDESTINATIONSTATE
D3D10_MESSAGE_ID_COPYRESOURCE_INVALIDSOURCESTATE
D3D10_MESSAGE_ID_UPDATESUBRESOURCE_INVALIDDESTINATIONSUBRESOURCE
D3D10_MESSAGE_ID_UPDATESUBRESOURCE_INVALIDDESTINATIONBOX
D3D10_MESSAGE_ID_UPDATESUBRESOURCE_INVALIDDESTINATIONSTATE
D3D10_MESSAGE_ID_DEVICE_RESOLVESUBRESOURCE_DESTINATION_INVALID
D3D10_MESSAGE_ID_DEVICE_RESOLVESUBRESOURCE_DESTINATION_SUBRESOURCE_INVALID
D3D10_MESSAGE_ID_DEVICE_RESOLVESUBRESOURCE_SOURCE_INVALID
D3D10_MESSAGE_ID_DEVICE_RESOLVESUBRESOURCE_SOURCE_SUBRESOURCE_INVALID
D3D10_MESSAGE_ID_DEVICE_RESOLVESUBRESOURCE_FORMAT_INVALID
D3D10_MESSAGE_ID_BUFFER_MAP_INVALIDMAPTYPE
D3D10_MESSAGE_ID_BUFFER_MAP_INVALIDFLAGS
D3D10_MESSAGE_ID_BUFFER_MAP_ALREADYMAPPED
D3D10_MESSAGE_ID_BUFFER_MAP_DEVICEREMOVED_RETURN

D3D10_MESSAGE_ID_BUFFER_UNMAP_NOTMAPPED
D3D10_MESSAGE_ID_TEXTURE1D_MAP_INVALIDMAPTYPE
D3D10_MESSAGE_ID_TEXTURE1D_MAP_INVALIDSUBRESOURCE
D3D10_MESSAGE_ID_TEXTURE1D_MAP_INVALIDFLAGS
D3D10_MESSAGE_ID_TEXTURE1D_MAP_ALREADYMAPPED
D3D10_MESSAGE_ID_TEXTURE1D_MAP_DEVICEREMOVED_RETURN
D3D10_MESSAGE_ID_TEXTURE1D_UNMAP_INVALIDSUBRESOURCE
D3D10_MESSAGE_ID_TEXTURE1D_UNMAP_NOTMAPPED
D3D10_MESSAGE_ID_TEXTURE2D_MAP_INVALIDMAPTYPE
D3D10_MESSAGE_ID_TEXTURE2D_MAP_INVALIDSUBRESOURCE
D3D10_MESSAGE_ID_TEXTURE2D_MAP_INVALIDFLAGS
D3D10_MESSAGE_ID_TEXTURE2D_MAP_ALREADYMAPPED
D3D10_MESSAGE_ID_TEXTURE2D_MAP_DEVICEREMOVED_RETURN
D3D10_MESSAGE_ID_TEXTURE2D_UNMAP_INVALIDSUBRESOURCE
D3D10_MESSAGE_ID_TEXTURE2D_UNMAP_NOTMAPPED
D3D10_MESSAGE_ID_TEXTURE3D_MAP_INVALIDMAPTYPE
D3D10_MESSAGE_ID_TEXTURE3D_MAP_INVALIDSUBRESOURCE
D3D10_MESSAGE_ID_TEXTURE3D_MAP_INVALIDFLAGS
D3D10_MESSAGE_ID_TEXTURE3D_MAP_ALREADYMAPPED
D3D10_MESSAGE_ID_TEXTURE3D_MAP_DEVICEREMOVED_RETURN
D3D10_MESSAGE_ID_TEXTURE3D_UNMAP_INVALIDSUBRESOURCE
D3D10_MESSAGE_ID_TEXTURE3D_UNMAP_NOTMAPPED
D3D10_MESSAGE_ID_CHECKFORMATSUPPORT_FORMAT_DEPRECATED
D3D10_MESSAGE_ID_CHECKMULTISAMPLEQUALITYLEVELS_FORMAT_DEPRECATED
D3D10_MESSAGE_ID_SETEXCEPTIONMODE_UNRECOGNIZEDFLAGS

D3D10_MESSAGE_ID_SETEXCEPTIONMODE_INVALIDARG_RETURN
D3D10_MESSAGE_ID_SETEXCEPTIONMODE_DEVICEREMOVED_RETURN
D3D10_MESSAGE_ID_REF_SIMULATING_INFINITELY_FAST_HARDWARE
D3D10_MESSAGE_ID_REF_THREADING_MODE
D3D10_MESSAGE_ID_REF_UMDRIVER_EXCEPTION
D3D10_MESSAGE_ID_REF_KMDRIVER_EXCEPTION
D3D10_MESSAGE_ID_REF_HARDWARE_EXCEPTION
D3D10_MESSAGE_ID_REF_ACCESSING_INDEXABLE_TEMP_OUT_OF_RANGE
D3D10_MESSAGE_ID_REF_PROBLEM_PARSING_SHADER
D3D10_MESSAGE_ID_REF_OUT_OF_MEMORY
D3D10_MESSAGE_ID_REF_INFO
D3D10_MESSAGE_ID_DEVICE_DRAW_VERTEXPOS_OVERFLOW
D3D10_MESSAGE_ID_DEVICE_DRAWINDEXED_INDEXPOS_OVERFLOW
D3D10_MESSAGE_ID_DEVICE_DRAWINSTANCED_VERTEXPOS_OVERFLOW
D3D10_MESSAGE_ID_DEVICE_DRAWINSTANCED_INSTANCEPOS_OVERFLOW
D3D10_MESSAGE_ID_DEVICE_DRAWINDEXEDINSTANCED_INSTANCEPOS_OVERFLOW
D3D10_MESSAGE_ID_DEVICE_DRAWINDEXEDINSTANCED_INDEXPOS_OVERFLOW
D3D10_MESSAGE_ID_DEVICE_DRAW_VERTEX_SHADER_NOT_SET
D3D10_MESSAGE_ID_DEVICE_SHADER_LINKAGE_SEMANTICNAME_NOT_FOUND
D3D10_MESSAGE_ID_DEVICE_SHADER_LINKAGE_REGISTERINDEX
D3D10_MESSAGE_ID_DEVICE_SHADER_LINKAGE_COMPONENTTYPE
D3D10_MESSAGE_ID_DEVICE_SHADER_LINKAGE_REGISTERMASK
D3D10_MESSAGE_ID_DEVICE_SHADER_LINKAGE_SYSTEMVALUE
D3D10_MESSAGE_ID_DEVICE_SHADER_LINKAGE_NEVERWRITTEN_ALWAYSREADS
D3D10_MESSAGE_ID_DEVICE_DRAW_VERTEX_BUFFER_NOT_SET

D3D10_MESSAGE_ID_DEVICE_DRAW_INPUTLAYOUT_NOT_SET
D3D10_MESSAGE_ID_DEVICE_DRAW_CONSTANT_BUFFER_NOT_SET
D3D10_MESSAGE_ID_DEVICE_DRAW_CONSTANT_BUFFER_TOO_SMALL
D3D10_MESSAGE_ID_DEVICE_DRAW_SAMPLER_NOT_SET
D3D10_MESSAGE_ID_DEVICE_DRAW_SHADERRESOURCEVIEW_NOT_SET
D3D10_MESSAGE_ID_DEVICE_DRAW_VIEW_DIMENSION_MISMATCH
D3D10_MESSAGE_ID_DEVICE_DRAW_VERTEX_BUFFER_STRIDE_TOO_SMALL
D3D10_MESSAGE_ID_DEVICE_DRAW_VERTEX_BUFFER_TOO_SMALL
D3D10_MESSAGE_ID_DEVICE_DRAW_INDEX_BUFFER_NOT_SET
D3D10_MESSAGE_ID_DEVICE_DRAW_INDEX_BUFFER_FORMAT_INVALID
D3D10_MESSAGE_ID_DEVICE_DRAW_INDEX_BUFFER_TOO_SMALL
D3D10_MESSAGE_ID_DEVICE_DRAW_GS_INPUT_PRIMITIVE_MISMATCH
D3D10_MESSAGE_ID_DEVICE_DRAW_RESOURCE_RETURN_TYPE_MISMATCH
D3D10_MESSAGE_ID_DEVICE_DRAW_POSITION_NOT_PRESENT
D3D10_MESSAGE_ID_DEVICE_DRAW_OUTPUT_STREAM_NOT_SET
D3D10_MESSAGE_ID_DEVICE_DRAW_BOUND_RESOURCE_MAPPED
D3D10_MESSAGE_ID_DEVICE_DRAW_INVALID_PRIMITIVETOPOLOGY
D3D10_MESSAGE_ID_DEVICE_DRAW_VERTEX_OFFSET_UNALIGNED
D3D10_MESSAGE_ID_DEVICE_DRAW_VERTEX_STRIDE_UNALIGNED
D3D10_MESSAGE_ID_DEVICE_DRAW_INDEX_OFFSET_UNALIGNED
D3D10_MESSAGE_ID_DEVICE_DRAW_OUTPUT_STREAM_OFFSET_UNALIGNED
D3D10_MESSAGE_ID_DEVICE_DRAW_RESOURCE_FORMAT_LD_UNSUPPORTED
D3D10_MESSAGE_ID_DEVICE_DRAW_RESOURCE_FORMAT_SAMPLE_UNSUPPORTED
D3D10_MESSAGE_ID_DEVICE_DRAW_RESOURCE_FORMAT_SAMPLE_C_UNSUPPORTED
D3D10_MESSAGE_ID_DEVICE_DRAW_RESOURCE_MULTISAMPLE_UNSUPPORTED

D3D10_MESSAGE_ID_DEVICE_DRAW_SO_TARGETS_BOUND_WITHOUT_SOURCE
D3D10_MESSAGE_ID_DEVICE_DRAW_SO_STRIDE_LARGER_THAN_BUFFER
D3D10_MESSAGE_ID_DEVICE_DRAW_OM_RENDER_TARGET_DOES_NOT_SUPPORT_BLENDING
D3D10_MESSAGE_ID_DEVICE_DRAW_OM_DUAL_SOURCE_BLENDING_CAN_ONLY_HAVE_RENDER_TARGET_0
D3D10_MESSAGE_ID_DEVICE_REMOVAL_PROCESS_AT_FAULT
D3D10_MESSAGE_ID_DEVICE_REMOVAL_PROCESS_POSSIBLY_AT_FAULT
D3D10_MESSAGE_ID_DEVICE_REMOVAL_PROCESS_NOT_AT_FAULT
D3D10_MESSAGE_ID_DEVICE_OPEN_SHARED_RESOURCE_INVALIDARG_RETURN
D3D10_MESSAGE_ID_DEVICE_OPEN_SHARED_RESOURCE_OUTOFMEMORY_RETURN
D3D10_MESSAGE_ID_DEVICE_OPEN_SHARED_RESOURCE_BADINTERFACE_RETURN
D3D10_MESSAGE_ID_DEVICE_DRAW_VIEWPORT_NOT_SET
D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_TRAILING_DIGIT_IN_SEMANTIC
D3D10_MESSAGE_ID_CREATEGEOMETRYSHADERWITHSTREAMOUTPUT_TRAILING_DIGIT_IN_SEMANTIC
D3D10_MESSAGE_ID_DEVICE_RSSETVIEWPORTS_DENORMFLUSH
D3D10_MESSAGE_ID_OMSETRENDERTARGETS_INVALIDVIEW
D3D10_MESSAGE_ID_DEVICE_SETTEXTFILTERSIZE_INVALIDDIMENSIONS
D3D10_MESSAGE_ID_DEVICE_DRAW_SAMPLER_MISMATCH
D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_TYPE_MISMATCH
D3D10_MESSAGE_ID_BLENDSTATE_GETDESC_LEGACY
D3D10_MESSAGE_ID_SHADERRESOURCEVIEW_GETDESC_LEGACY
D3D10_MESSAGE_ID_CREATEQUERY_OUTOFMEMORY_RETURN
D3D10_MESSAGE_ID_CREATEPREDICATE_OUTOFMEMORY_RETURN
D3D10_MESSAGE_ID_CREATECOUNTER_OUTOFRANGE_COUNTER
D3D10_MESSAGE_ID_CREATECOUNTER_SIMULTANEOUS_ACTIVE_COUNTERS_EXHAUSTED
D3D10_MESSAGE_ID_CREATECOUNTER_UNSUPPORTED_WELLKNOWN_COUNTER

D3D10_MESSAGE_ID_CREATECOUNTER_OUTOFMEMORY_RETURN
D3D10_MESSAGE_ID_CREATECOUNTER_NONEXCLUSIVE_RETURN
D3D10_MESSAGE_ID_CREATECOUNTER_NULLDESC
D3D10_MESSAGE_ID_CHECKCOUNTER_OUTOFRANGE_COUNTER
D3D10_MESSAGE_ID_CHECKCOUNTER_UNSUPPORTED_WELLKNOWN_COUNTER
D3D10_MESSAGE_ID_SETPREDICATION_INVALID_PREDICATE_STATE
D3D10_MESSAGE_ID_QUERY_BEGIN_UNSUPPORTED
D3D10_MESSAGE_ID_PREDICATE_BEGIN_DURING_PREDICATION
D3D10_MESSAGE_ID_QUERY_BEGIN_DUPLICATE
D3D10_MESSAGE_ID_QUERY_BEGIN_ABANDONING_PREVIOUS_RESULTS
D3D10_MESSAGE_ID_PREDICATE_END_DURING_PREDICATION
D3D10_MESSAGE_ID_QUERY_END_ABANDONING_PREVIOUS_RESULTS
D3D10_MESSAGE_ID_QUERY_END_WITHOUT_BEGIN
D3D10_MESSAGE_ID_QUERY_GETDATA_INVALID_DATASIZE
D3D10_MESSAGE_ID_QUERY_GETDATA_INVALID_FLAGS
D3D10_MESSAGE_ID_QUERY_GETDATA_INVALID_CALL
D3D10_MESSAGE_ID_DEVICE_DRAW_PS_OUTPUT_TYPE_MISMATCH
D3D10_MESSAGE_ID_DEVICE_DRAW_RESOURCE_FORMAT_GATHER_UNSUPPORTED
D3D10_MESSAGE_ID_DEVICE_DRAW_INVALID_USE_OF_CENTER_MULTISAMPLE_PATTERN
D3D10_MESSAGE_ID_DEVICE_IASETVERTEXBUFFERS_STRIDE_TOO_LARGE
D3D10_MESSAGE_ID_DEVICE_IASETVERTEXBUFFERS_INVALIDRANGE
D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_EMPTY_LAYOUT
D3D10_MESSAGE_ID_DEVICE_DRAW_RESOURCE_SAMPLE_COUNT_MISMATCH
D3D10_MESSAGE_ID_LIVE_OBJECT_SUMMARY
D3D10_MESSAGE_ID_LIVE_BUFFER

`D3D10_MESSAGE_ID_LIVE_TEXTURE1D`
`D3D10_MESSAGE_ID_LIVE_TEXTURE2D`
`D3D10_MESSAGE_ID_LIVE_TEXTURE3D`
`D3D10_MESSAGE_ID_LIVE_SHADERRESOURCEVIEW`
`D3D10_MESSAGE_ID_LIVE_RENDERTARGETVIEW`
`D3D10_MESSAGE_ID_LIVE_DEPTHSTENCILVIEW`
`D3D10_MESSAGE_ID_LIVE_VERTEXSHADER`
`D3D10_MESSAGE_ID_LIVE_GEOMETRYSHADER`
`D3D10_MESSAGE_ID_LIVE_PIXELSHADER`
`D3D10_MESSAGE_ID_LIVE_INPUTLAYOUT`
`D3D10_MESSAGE_ID_LIVE_SAMPLER`
`D3D10_MESSAGE_ID_LIVE_BLENDSTATE`
`D3D10_MESSAGE_ID_LIVE_DEPTHSTENCILSTATE`
`D3D10_MESSAGE_ID_LIVE_RASTERIZERSTATE`
`D3D10_MESSAGE_ID_LIVE_QUERY`
`D3D10_MESSAGE_ID_LIVE_PREDICATE`
`D3D10_MESSAGE_ID_LIVE_COUNTER`
`D3D10_MESSAGE_ID_LIVE_DEVICE`
`D3D10_MESSAGE_ID_LIVE_SWAPCHAIN`
`D3D10_MESSAGE_ID_D3D10_MESSAGES_END`
`D3D10_MESSAGE_ID_D3D10L9_MESSAGES_START` Value: *0x100000*
`D3D10_MESSAGE_ID_CREATEDEPTHSTENCILSTATE_STENCIL_NO_TWO_SIDED`
`D3D10_MESSAGE_ID_CREATERASTERIZERSTATE_DepthBiasClamp_NOT_SUPPORTED`
`D3D10_MESSAGE_ID_CREATESAMPLERSTATE_NO_COMPARISON_SUPPORT`
`D3D10_MESSAGE_ID_CREATESAMPLERSTATE_EXCESSIVE_ANISOTROPY`

D3D10_MESSAGE_ID_CREATESAMPLERSTATE_BORDER_OUT_OF_RANGE
D3D10_MESSAGE_ID_VSSETSAMPLERS_NOT_SUPPORTED
D3D10_MESSAGE_ID_VSSETSAMPLERS_TOO_MANY_SAMPLERS
D3D10_MESSAGE_ID_PSSETSAMPLERS_TOO_MANY_SAMPLERS
D3D10_MESSAGE_ID_CREATERESOURCE_NO_ARRAYS
D3D10_MESSAGE_ID_CREATERESOURCE_NO_VB_AND_IB_BIND
D3D10_MESSAGE_ID_CREATERESOURCE_NO_TEXTURE_1D
D3D10_MESSAGE_ID_CREATERESOURCE_DIMENSION_OUT_OF_RANGE
D3D10_MESSAGE_ID_CREATERESOURCE_NOT_BINDABLE_AS_SHADER_RESOURCE
D3D10_MESSAGE_ID_OMSETRENDERTARGETS_TOO_MANY_RENDER_TARGETS
D3D10_MESSAGE_ID_OMSETRENDERTARGETS_NO_DIFFERING_BIT_DEPTHS
D3D10_MESSAGE_ID_IASETVERTEXBUFFERS_BAD_BUFFER_INDEX
D3D10_MESSAGE_ID_DEVICE_RSSETVIEWPORTS_TOO_MANY_VIEWPORTS
D3D10_MESSAGE_ID_DEVICE_IASETPRIMITIVETOPOLOGY_ADJACENCY_UNSUPPORTED
D3D10_MESSAGE_ID_DEVICE_RSSETSCISSORRECTS_TOO_MANY_SCISSORS
D3D10_MESSAGE_ID_COPYRESOURCE_ONLY_TEXTURE_2D_WITHIN_GPU_MEMORY
D3D10_MESSAGE_ID_COPYRESOURCE_NO_TEXTURE_3D_READBACK
D3D10_MESSAGE_ID_COPYRESOURCE_NO_TEXTURE_ONLY_READBACK
D3D10_MESSAGE_ID_CREATEINPUTLAYOUT_UNSUPPORTED_FORMAT
D3D10_MESSAGE_ID_CREATEBLENDSTATE_NO_ALPHA_TO_COVERAGE
D3D10_MESSAGE_ID_CREATERASTERIZERSTATE_DepthClipEnable_MUST_BE_TRUE
D3D10_MESSAGE_ID_DRAWINDEXED_STARTINDEXLOCATION_MUST_BE_POSITIVE
D3D10_MESSAGE_ID_CREATESHADERRESOURCEVIEW_MUST_USE_LOWEST_LOD
D3D10_MESSAGE_ID_CREATESAMPLERSTATE_MINLOD_MUST_NOT_BE_FRACTIONAL
D3D10_MESSAGE_ID_CREATESAMPLERSTATE_MAXLOD_MUST_BE_FLT_MAX

D3D10_MESSAGE_ID_CREATESHADERRESOURCEVIEW_FIRSTARRAYSLICE_MUST_BE_ZERO
D3D10_MESSAGE_ID_CREATESHADERRESOURCEVIEW_CUBES_MUST_HAVE_6_SIDES
D3D10_MESSAGE_ID_CREATERESOURCE_NOT_BINDABLE_AS_RENDER_TARGET
D3D10_MESSAGE_ID_CREATERESOURCE_NO_DWORD_INDEX_BUFFER
D3D10_MESSAGE_ID_CREATERESOURCE_MSAA_PRECLUDES_SHADER_RESOURCE
D3D10_MESSAGE_ID_CREATERESOURCE_PRESENTATION_PRECLUDES_SHADER_RESOURCE
D3D10_MESSAGE_ID_CREATEBLENDSTATE_NO_INDEPENDENT_BLEND_ENABLE
D3D10_MESSAGE_ID_CREATEBLENDSTATE_NO_INDEPENDENT_WRITE_MASKS
D3D10_MESSAGE_ID_CREATERESOURCE_NO_STREAM_OUT
D3D10_MESSAGE_ID_CREATERESOURCE_ONLY_VB_IB_FOR_BUFFERS
D3D10_MESSAGE_ID_CREATERESOURCE_NO_AUTOGEN_FOR_VOLUMES
D3D10_MESSAGE_ID_CREATERESOURCE_DXGI_FORMAT_R8G8B8A8_CANNOT_BE_SHARED
D3D10_MESSAGE_ID_VSSHADERRESOURCES_NOT_SUPPORTED
D3D10_MESSAGE_ID_GEOMETRY_SHADER_NOT_SUPPORTED
D3D10_MESSAGE_ID_STREAM_OUT_NOT_SUPPORTED
D3D10_MESSAGE_ID_TEXT_FILTER_NOT_SUPPORTED
D3D10_MESSAGE_ID_CREATEBLENDSTATE_NO_SEPARATE_ALPHA_BLEND
D3D10_MESSAGE_ID_CREATEBLENDSTATE_NO_MRT_BLEND
D3D10_MESSAGE_ID_CREATEBLENDSTATE_OPERATION_NOT_SUPPORTED
D3D10_MESSAGE_ID_CREATESAMPLERSTATE_NO_MIRRORONCE
D3D10_MESSAGE_ID_DRAWINSTANCED_NOT_SUPPORTED
D3D10_MESSAGE_ID_DRAWINDEXEDINSTANCED_NOT_SUPPORTED_BELOW_9_3
D3D10_MESSAGE_ID_DRAWINDEXED_POINTLIST_UNSUPPORTED
D3D10_MESSAGE_ID_SETBLENDSTATE_SAMPLE_MASK_CANNOT_BE_ZERO
D3D10_MESSAGE_ID_CREATERESOURCE_DIMENSION_EXCEEDS_FEATURE_LEVEL_DEFINITION

`D3D10_MESSAGE_ID_CREATERESOURCE_ONLY_SINGLE_MIP_LEVEL_DEPTH_STENCIL_SUPPORTED`
`D3D10_MESSAGE_ID_DEVICE_RSSETSCISSORRECTS_NEGATIVESCISSOR`
`D3D10_MESSAGE_ID_SLOT_ZERO_MUST_BE_D3D10_INPUT_PER_VERTEX_DATA`
`D3D10_MESSAGE_ID_CREATERESOURCE_NON_POW_2_MIPMAP`
`D3D10_MESSAGE_ID_CREATESAMPLERSTATE_BORDER_NOT_SUPPORTED`
`D3D10_MESSAGE_ID_OMSETRENDERTARGETS_NO_SRGB_MRT`
`D3D10_MESSAGE_ID_COPYRESOURCE_NO_3D_MISMATCHED_UPDATES`
`D3D10_MESSAGE_ID_D3D10L9_MESSAGES_END`

Requirements

Header	d3d10sdklayers.h

See also

Core Enumerations

D3D10_MESSAGE_SEVERITY enumeration (d3d10sdklayers.h)

Article07/27/2022

Debug message severity levels for an information queue.

Syntax

C++

```
typedef enum D3D10_MESSAGE_SEVERITY {
  D3D10_MESSAGE_SEVERITY_CORRUPTION = 0,
  D3D10_MESSAGE_SEVERITY_ERROR,
  D3D10_MESSAGE_SEVERITY_WARNING,
  D3D10_MESSAGE_SEVERITY_INFO,
  D3D10_MESSAGE_SEVERITY_MESSAGE
} ;
```

Constants

`D3D10_MESSAGE_SEVERITY_CORRUPTION` Value: *0* Defines some type of corruption which has occurred.
`D3D10_MESSAGE_SEVERITY_ERROR` Defines an error message.
`D3D10_MESSAGE_SEVERITY_WARNING` Defines a warning message.
`D3D10_MESSAGE_SEVERITY_INFO` Defines an information message.
`D3D10_MESSAGE_SEVERITY_MESSAGE`

Remarks

Use these values to allow or deny message categories to pass through the storage and retrieval filters for an information queue (see D3D10_INFO_QUEUE_FILTER). This API is used by ID3D10InfoQueue::AddApplicationMessage.

Requirements

Header	d3d10sdklayers.h (include D3D10.h)

See also

Core Enumerations

D3D10_QUERY enumeration (d3d10.h)

Article • 07/27/20222 minutes to read

Query types.

Syntax

C++

```
typedef enum D3D10_QUERY {
  D3D10_QUERY_EVENT = 0,
  D3D10_QUERY_OCCLUSION,
  D3D10_QUERY_TIMESTAMP,
  D3D10_QUERY_TIMESTAMP_DISJOINT,
  D3D10_QUERY_PIPELINE_STATISTICS,
  D3D10_QUERY_OCCLUSION_PREDICATE,
  D3D10_QUERY_SO_STATISTICS,
  D3D10_QUERY_SO_OVERFLOW_PREDICATE
} ;
```

Constants

`D3D10_QUERY_EVENT` Value: *0* Determines whether or not the GPU is finished processing commands. When the GPU is finished processing commands GetData will return S_OK, and pData will point to a BOOL with a value of **TRUE**. When using this type of query, Begin is disabled.
`D3D10_QUERY_OCCLUSION` Get the number of samples that passed the depth and stencil tests in between Begin and End. GetData returns a UINT64. If a depth or stencil test is disabled, then each of those tests will be counted as a pass.
`D3D10_QUERY_TIMESTAMP` Get a timestamp value where GetData returns a UINT64. This kind of query is only useful if two timestamp queries are done in the middle of a D3D10_QUERY_TIMESTAMP_DISJOINT query. The difference of two timestamps can be used to determine how many ticks have elapsed, and the D3D10_QUERY_TIMESTAMP_DISJOINT query will determine if that difference is a reliable value and also has a value that shows how to convert the number of ticks into seconds. See D3D10_QUERY_DATA_TIMESTAMP_DISJOINT. When using this type of query, Begin is disabled.

`D3D10_QUERY_TIMESTAMP_DISJOINT`	Determines whether or not a D3D10_QUERY_TIMESTAMP is returning reliable values, and also gives the frequency of the processor enabling you to convert the number of elapsed ticks into seconds. GetData will return a D3D10_QUERY_DATA_TIMESTAMP_DISJOINT. This type of query should only be invoked once per frame or less.
`D3D10_QUERY_PIPELINE_STATISTICS`	Get pipeline statistics, such as the number of pixel shader invocations in between Begin and End. GetData will return a D3D10_QUERY_DATA_PIPELINE_STATISTICS.
`D3D10_QUERY_OCCLUSION_PREDICATE`	Similar to D3D10_QUERY_OCCLUSION, except GetData returns a BOOL indicating whether or not any samples passed the depth and stencil tests - **TRUE** meaning at least one passed, **FALSE** meaning none passed.
`D3D10_QUERY_SO_STATISTICS`	Get streaming output statistics, such as the number of primitives streamed out in between Begin and End. GetData will return a D3D10_QUERY_DATA_SO_STATISTICS structure.
`D3D10_QUERY_SO_OVERFLOW_PREDICATE`	Determines whether or not any of the streaming output buffers overflowed in between Begin and End. GetData returns a BOOL - **TRUE** meaning there was an overflow, **FALSE** meaning there was not an overflow. If streaming output writes to multiple buffers, and one of the buffers overflows, then it will stop writing to all the output buffers. When an overflow is detected by Direct3D it is prevented from happening - no memory is corrupted. This predication may be used in conjunction with an SO_STATISTICS query so that when an overflow occurs the SO_STATISTIC query will let the application know how much memory was needed to prevent an overflow.

Requirements

Header	d3d10.h

See also

Core Enumerations

D3D10_QUERY_MISC_FLAG enumeration (d3d10.h)

Article • 07/27/20222 minutes to read

Flags that describe miscellaneous query behavior.

Syntax

C++

```
typedef enum D3D10_QUERY_MISC_FLAG {
  D3D10_QUERY_MISC_PREDICATEHINT = 0x1
} ;
```

Constants

`D3D10_QUERY_MISC_PREDICATEHINT` Value: *0x1* Tell the hardware that if it is not yet sure if something is hidden or not to draw it anyway. This is only used with an occlusion predicate. Predication data cannot be returned to your application via ID3D10Asynchronous::GetData when using this flag.

Remarks

This flag is part of a query description (see D3D10_QUERY_DESC).

Requirements

Header	d3d10.h

See also

Core Enumerations

D3D10_RAISE_FLAG enumeration (d3d10.h)

Article • 07/27/20222 minutes to read

Option(s) for raising an error to a non-continuable exception.

Syntax

C++

```
typedef enum D3D10_RAISE_FLAG {
  D3D10_RAISE_FLAG_DRIVER_INTERNAL_ERROR = 0x1L
} ;
```

Constants

`D3D10_RAISE_FLAG_DRIVER_INTERNAL_ERROR` Value: *0x1L* Raise an internal driver error to a non-continuable exception.

Remarks

These flags are used by ID3D10Device::GetExceptionMode and ID3D10Device::SetExceptionMode. Use 0 to indicate no flags; multiple flags can be logically OR'ed together.

Requirements

Header	d3d10.h

See also

Core Enumerations

D3D_REGISTER_COMPONENT_TYPE enumeration (d3dcommon.h)

Article • 01/31/20222 minutes to read

Values that identify the data types that can be stored in a register.

> **Note**
>
> For programming with Direct3D 10, this API has a type alias that begins `D3D10_` instead of `D3D_`. These Direct3D 10 type aliases are defined in `d3d10.h`, `d3d10misc.h`, and `d3d10shader.h`.

Syntax

C++

```
typedef enum D3D_REGISTER_COMPONENT_TYPE {
  D3D_REGISTER_COMPONENT_UNKNOWN = 0,
  D3D_REGISTER_COMPONENT_UINT32 = 1,
  D3D_REGISTER_COMPONENT_SINT32 = 2,
  D3D_REGISTER_COMPONENT_FLOAT32 = 3,
  D3D10_REGISTER_COMPONENT_UNKNOWN,
  D3D10_REGISTER_COMPONENT_UINT32,
  D3D10_REGISTER_COMPONENT_SINT32,
  D3D10_REGISTER_COMPONENT_FLOAT32
} ;
```

Constants

`D3D_REGISTER_COMPONENT_UNKNOWN` Value: *0* The data type is unknown.
`D3D_REGISTER_COMPONENT_UINT32` Value: *1* 32-bit unsigned integer.

`D3D_REGISTER_COMPONENT_SINT32` Value: *2* 32-bit signed integer.	
`D3D_REGISTER_COMPONENT_FLOAT32` Value: *3* 32-bit floating-point number.	
`D3D10_REGISTER_COMPONENT_UNKNOWN` The data type is unknown.	
`D3D10_REGISTER_COMPONENT_UINT32` 32-bit unsigned integer.	
`D3D10_REGISTER_COMPONENT_SINT32` 32-bit signed integer.	
`D3D10_REGISTER_COMPONENT_FLOAT32` 32-bit floating-point number.	

Remarks

A register component type is specified in the **ComponentType** member of the D3D11_SIGNATURE_PARAMETER_DESC structure.

Requirements

Header	d3dcommon.h

See also

Common Version Enumerations

D3D_RESOURCE_RETURN_TYPE enumeration (d3dcommon.h)

Article • 01/31/20222 minutes to read

Indicates return value type.

> **Note**
>
> For programming with Direct3D 10, this API has a type alias that begins `D3D10_` instead of `D3D_`. These Direct3D 10 type aliases are defined in `d3d10.h`, `d3d10misc.h`, and `d3d10shader.h`.

Syntax

C++

```
typedef enum D3D_RESOURCE_RETURN_TYPE {
  D3D_RETURN_TYPE_UNORM = 1,
  D3D_RETURN_TYPE_SNORM = 2,
  D3D_RETURN_TYPE_SINT = 3,
  D3D_RETURN_TYPE_UINT = 4,
  D3D_RETURN_TYPE_FLOAT = 5,
  D3D_RETURN_TYPE_MIXED = 6,
  D3D_RETURN_TYPE_DOUBLE = 7,
  D3D_RETURN_TYPE_CONTINUED = 8,
  D3D10_RETURN_TYPE_UNORM,
  D3D10_RETURN_TYPE_SNORM,
  D3D10_RETURN_TYPE_SINT,
  D3D10_RETURN_TYPE_UINT,
  D3D10_RETURN_TYPE_FLOAT,
  D3D10_RETURN_TYPE_MIXED,
  D3D11_RETURN_TYPE_UNORM,
  D3D11_RETURN_TYPE_SNORM,
  D3D11_RETURN_TYPE_SINT,
  D3D11_RETURN_TYPE_UINT,
  D3D11_RETURN_TYPE_FLOAT,
  D3D11_RETURN_TYPE_MIXED,
  D3D11_RETURN_TYPE_DOUBLE,
  D3D11_RETURN_TYPE_CONTINUED
} ;
```

Constants

`D3D_RETURN_TYPE_UNORM` Value: *1*
`D3D_RETURN_TYPE_SNORM` Value: *2*
`D3D_RETURN_TYPE_SINT` Value: *3*
`D3D_RETURN_TYPE_UINT` Value: *4*
`D3D_RETURN_TYPE_FLOAT` Value: *5*
`D3D_RETURN_TYPE_MIXED` Value: *6*
`D3D_RETURN_TYPE_DOUBLE` Value: *7*
`D3D_RETURN_TYPE_CONTINUED` Value: *8*
`D3D10_RETURN_TYPE_UNORM`
`D3D10_RETURN_TYPE_SNORM`
`D3D10_RETURN_TYPE_SINT`
`D3D10_RETURN_TYPE_UINT`
`D3D10_RETURN_TYPE_FLOAT`
`D3D10_RETURN_TYPE_MIXED`
`D3D11_RETURN_TYPE_UNORM` Return type is UNORM.
`D3D11_RETURN_TYPE_SNORM` Return type is SNORM.
`D3D11_RETURN_TYPE_SINT` Return type is SINT.
`D3D11_RETURN_TYPE_UINT` Return type is UINT.

`D3D11_RETURN_TYPE_FLOAT`	Return type is FLOAT.
`D3D11_RETURN_TYPE_MIXED`	Return type is unknown.
`D3D11_RETURN_TYPE_DOUBLE`	Return type is DOUBLE.
`D3D11_RETURN_TYPE_CONTINUED`	Return type is a multiple-dword type, such as a double or uint64, and the component is continued from the previous component that was declared. The first component represents the lower bits.

Remarks

The **D3D11_RESOURCE_RETURN_TYPE** enumeration is type defined in the D3D11shader.h header file as a D3D_RESOURCE_RETURN_TYPE enumeration, which is fully defined in the D3DCommon.h header file.

```
typedef D3D_RESOURCE_RETURN_TYPE D3D11_RESOURCE_RETURN_TYPE;
```

Requirements

Header	d3dcommon.h

See also

Shader Enumerations

D3D10_STENCIL_OP enumeration (d3d10.h)

Article • 07/27/20222 minutes to read

The stencil operations that can be performed during depth-stencil testing.

Syntax

C++

```
typedef enum D3D10_STENCIL_OP {
  D3D10_STENCIL_OP_KEEP = 1,
  D3D10_STENCIL_OP_ZERO = 2,
  D3D10_STENCIL_OP_REPLACE = 3,
  D3D10_STENCIL_OP_INCR_SAT = 4,
  D3D10_STENCIL_OP_DECR_SAT = 5,
  D3D10_STENCIL_OP_INVERT = 6,
  D3D10_STENCIL_OP_INCR = 7,
  D3D10_STENCIL_OP_DECR = 8
} ;
```

Constants

`D3D10_STENCIL_OP_KEEP` Value: *1* Keep the existing stencil data.
`D3D10_STENCIL_OP_ZERO` Value: *2* Set the stencil data to 0.
`D3D10_STENCIL_OP_REPLACE` Value: *3* Set the stencil data to the reference value set by calling ID3D10Device::OMSetDepthStencilState.
`D3D10_STENCIL_OP_INCR_SAT` Value: *4* Increment the stencil value by 1, and clamp the result.
`D3D10_STENCIL_OP_DECR_SAT` Value: *5* Decrement the stencil value by 1, and clamp the result.

`D3D10_STENCIL_OP_INVERT` Value: *6* Invert the stencil data.
`D3D10_STENCIL_OP_INCR` Value: *7* Increment the stencil value by 1, and wrap the result if necessary.
`D3D10_STENCIL_OP_DECR` Value: *8* Increment the stencil value by 1, and wrap the result if necessary.

Requirements

Header	d3d10.h

See also

Core Enumerations

D3D10_TEXTURECUBE_FACE enumeration (d3d10.h)

Article • 07/27/20222 minutes to read

The different faces of a cube texture.

Syntax

C++

```
typedef enum D3D10_TEXTURECUBE_FACE {
  D3D10_TEXTURECUBE_FACE_POSITIVE_X = 0,
  D3D10_TEXTURECUBE_FACE_NEGATIVE_X = 1,
  D3D10_TEXTURECUBE_FACE_POSITIVE_Y = 2,
  D3D10_TEXTURECUBE_FACE_NEGATIVE_Y = 3,
  D3D10_TEXTURECUBE_FACE_POSITIVE_Z = 4,
  D3D10_TEXTURECUBE_FACE_NEGATIVE_Z = 5
} ;
```

Constants

`D3D10_TEXTURECUBE_FACE_POSITIVE_X` Value: *0* Positive X face.
`D3D10_TEXTURECUBE_FACE_NEGATIVE_X` Value: *1* Negative X face.
`D3D10_TEXTURECUBE_FACE_POSITIVE_Y` Value: *2* Positive Y face.
`D3D10_TEXTURECUBE_FACE_NEGATIVE_Y` Value: *3* Negative Y face.
`D3D10_TEXTURECUBE_FACE_POSITIVE_Z` Value: *4* Positive Z face.

`D3D10_TEXTURECUBE_FACE_NEGATIVE_Z` Value: *5* Negative Z face.	

Requirements

Header	d3d10.h

See also

Core Enumerations

D3D10_TEXTURE_ADDRESS_MODE enumeration (d3d10.h)

Article07/27/2022

Identify a technique for resolving texture coordinates that are outside of the boundaries of a texture.

Syntax

C++

```
typedef enum D3D10_TEXTURE_ADDRESS_MODE {
  D3D10_TEXTURE_ADDRESS_WRAP = 1,
  D3D10_TEXTURE_ADDRESS_MIRROR = 2,
  D3D10_TEXTURE_ADDRESS_CLAMP = 3,
  D3D10_TEXTURE_ADDRESS_BORDER = 4,
  D3D10_TEXTURE_ADDRESS_MIRROR_ONCE = 5
} ;
```

Constants

`D3D10_TEXTURE_ADDRESS_WRAP` Value: *1* Tile the texture at every integer junction. For example, for u values between 0 and 3, the texture is repeated three times.
`D3D10_TEXTURE_ADDRESS_MIRROR` Value: *2* Flip the texture at every integer junction. For u values between 0 and 1, for example, the texture is addressed normally; between 1 and 2, the texture is flipped (mirrored); between 2 and 3, the texture is normal again; and so on.
`D3D10_TEXTURE_ADDRESS_CLAMP` Value: *3* Texture coordinates outside the range [0.0, 1.0] are set to the texture color at 0.0 or 1.0, respectively.
`D3D10_TEXTURE_ADDRESS_BORDER` Value: *4* Texture coordinates outside the range [0.0, 1.0] are set to the border color specified in D3D10_SAMPLER_DESC or HLSL code.

`D3D10_TEXTURE_ADDRESS_MIRROR_ONCE` Value: *5* Similar to D3D10_TEXTURE_ADDRESS_MIRROR and D3D10_TEXTURE_ADDRESS_CLAMP. Takes the absolute value of the texture coordinate (thus, mirroring around 0), and then clamps to the maximum value.

Requirements

Header	d3d10.h

See also

Core Enumerations

Resource Reference (Direct3D 10 Graphics)

Article • 01/06/2021 • 2 minutes to read

The Direct3D API defines several API elements to help you create and manage resources:

- Resource Interfaces
- Resource Functions
- Resource Structures
- Resource Enumerations
- Resource Constants

Related topics

Direct3D Reference

Resource Interfaces (Direct3D 10 Graphics)

Article • 01/06/2021 • 2 minutes to read

Direct3D 10 defines a number of interfaces for the two basic types of resources: buffers and textures.

Interfaces	Description
ID3D10Buffer Interface	Accesses buffer data.
ID3D10Resource Interface	Base class for a resource.
ID3D10Texture1D Interface	Accesses data in a 1D texture or a 1D texture array.
ID3D10Texture2D Interface	Accesses data in a 2D texture or a 2D texture array
ID3D10Texture3D Interface	Accesses data in a 3D texture or a 3D texture array.

An application uses a view to bind a resource to a pipeline stage. The view defines how the resource can be accessed during rendering. The API contains these view interfaces.

Interfaces	Description
ID3D10DepthStencilView Interface	Accesses data in a depth-stencil texture.
ID3D10RenderTargetView Interface	Accesses data in a render target.
ID3D10ShaderResourceView Interface	Accesses data in a shader-resource in Direct3D 10.0.
ID3D10ShaderResourceView1 Interface	Accesses data in a shader-resource in Direct3D 10.1.
ID3D10View Interface	Base class for a view.

Related topics

Resource Reference

ID3D10Buffer interface (d3d10.h)

Article • 07/27/20222 minutes to read

A buffer interface accesses a buffer resource, which is unstructured memory. Buffers typically store vertex or index data.

Inheritance

The **ID3D10Buffer** interface inherits from ID3D10Resource. **ID3D10Buffer** also has these types of members:

Methods

The **ID3D10Buffer** interface has these methods.

ID3D10Buffer::GetDesc Get the properties of a buffer resource. (ID3D10Buffer.GetDesc)
ID3D10Buffer::Map Get a pointer to the data contained in the resource and deny GPU access to the resource.
ID3D10Buffer::Unmap Invalidate the pointer to the resource retrieved by ID3D10Buffer::Map and reenable GPU access to the resource.

Remarks

Three types of buffers can be created; vertex, index, and shader-constant buffers. To create a buffer resource, call ID3D10Device::CreateBuffer.

A buffer must be bound to the pipeline before it can be accessed. Buffers can be bound to the input-assembler stage by calls to ID3D10Device::IASetVertexBuffers and ID3D10Device::IASetIndexBuffer, and to the stream-output stage by a call to ID3D10Device::SOSetTargets.

Buffers can be bound to multiple pipeline stages simultaneously for reading. A buffer can also be bound to a single pipeline stage for writing; however, the same buffer

cannot be bound for reading and writing simultaneously. For more information, see binding resources.

Requirements

Target Platform	Windows
Header	d3d10.h

See also

ID3D10Resource

Resource Interfaces

ID3D10Buffer::GetDesc method (d3d10.h)

Article07/27/2022

Get the properties of a buffer resource.

Syntax

```
C++

void GetDesc(
  [out] D3D10_BUFFER_DESC *pDesc
);
```

Parameters

`[out] pDesc`

Type: **D3D10_BUFFER_DESC***

Pointer to a resource description (see D3D10_BUFFER_DESC) filled in by the method. This pointer cannot be **NULL**.

Return value

None

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Buffer Interface

ID3D10Buffer::Map method (d3d10.h)

Article • 10/13/20212 minutes to read

Get a pointer to the data contained in the resource and deny GPU access to the resource.

Syntax

```
C++

HRESULT Map(
  [in]  D3D10_MAP MapType,
  [in]  UINT      MapFlags,
  [out] void      **ppData
);
```

Parameters

`[in] MapType`

Type: **D3D10_MAP**

Flag that specifies the CPU's permissions for the reading and writing of a resource. For possible values, see D3D10_MAP.

`[in] MapFlags`

Type: **UINT**

Flag that specifies what the CPU should do when the GPU is busy (see D3D10_MAP_FLAG). This flag is optional.

`[out] ppData`

Type: **void****

Pointer to the buffer resource data.

Return value

Type: **HRESULT**

If this function succeeds, it returns S_OK. The following list contains some of the reasons that **Map** can fail:

- If *MapFlags* specifies D3D10_MAP_FLAG_DO_NOT_WAIT and the GPU is not yet finished with the resource, **ID3D10Buffer::Map** returns DXGI_ERROR_WAS_STILL_DRAWING.
- **ID3D10Buffer::Map** returns DXGI_ERROR_DEVICE_REMOVED if *MapType* includes any flags that permit reading and the hardware device (that is, the video card) has been removed.

For more information about the preceding return values, see DXGI_ERROR.

Remarks

For the CPU to write the contents of a resource, the resource must be created with the dynamic usage flag, D3D10_USAGE_DYNAMIC. To both read and write those contents, the resource must be created with the staging usage flag, D3D10_USAGE_STAGING. (For more information about these flags, see D3D10_USAGE.) **ID3D10Buffer::Map** will retrieve a pointer to the resource data. For a discussion on how to access resources efficiently, see Copying and Accessing Resource Data (Direct3D 10).

Call ID3D10Buffer::Unmap to signify that the application has finished accessing the resource.

ID3D10Buffer::Map has a few other restrictions. For example:

- The same buffer cannot be mapped multiple times; in other words, do not call **ID3D10Buffer::Map** on a buffer that is already mapped.
- Any buffer that is bound to the pipeline must be unmapped before any rendering operation (that is, ID3D10Device::Draw) can be executed.

Differences between Direct3D 9 and Direct3D 10: **ID3D10Buffer::Map** in Direct3D 10 is analogous to resource Lock in Direct3D 9.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Buffer Interface

ID3D10Buffer::Unmap method (d3d10.h)

Article • 06/29/20212 minutes to read

Invalidate the pointer to the resource retrieved by ID3D10Buffer::Map and reenable GPU access to the resource.

Syntax

```
C++

void Unmap();
```

Return value

None

Remarks

Differences between Direct3D 9 and Direct3D 10:
Unmap() in Direct3D 10 is analogous to resource Unlock() in Direct3D 9.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Buffer Interface

ID3D10DepthStencilView interface (d3d10.h)

Article • 07/27/20222 minutes to read

A depth-stencil-view interface accesses a texture resource during depth-stencil testing.

Inheritance

The **ID3D10DepthStencilView** interface inherits from ID3D10View.
ID3D10DepthStencilView also has these types of members:

Methods

The **ID3D10DepthStencilView** interface has these methods.

ID3D10DepthStencilView::GetDesc Get the depth-stencil view. (ID3D10DepthStencilView.GetDesc)

Remarks

To create a depth-stencil view, call ID3D10Device::CreateDepthStencilView.

To bind a depth-stencil view to the pipeline, call ID3D10Device::OMSetRenderTargets.

Requirements

Target Platform	Windows
Header	d3d10.h

See also

ID3D10View

ID3D10DepthStencilView::GetDesc method (d3d10.h)

Article • 07/27/20222 minutes to read

Get the depth-stencil view.

Syntax

```
C++

void GetDesc(
  [out] D3D10_DEPTH_STENCIL_VIEW_DESC *pDesc
);
```

Parameters

`[out] pDesc`

Type: **D3D10_DEPTH_STENCIL_VIEW_DESC***

Pointer to a depth-stencil-view description (see D3D10_DEPTH_STENCIL_VIEW_DESC).

Return value

None

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10DepthStencilView Interface

ID3D10RenderTargetView interface (d3d10.h)

Article • 07/27/20222 minutes to read

A render-target-view interface identifies the render-target subresources that can be accessed during rendering.

Inheritance

The **ID3D10RenderTargetView** interface inherits from ID3D10View. **ID3D10RenderTargetView** also has these types of members:

Methods

The **ID3D10RenderTargetView** interface has these methods.

ID3D10RenderTargetView::GetDesc Get the properties of a render target view. (ID3D10RenderTargetView.GetDesc)

Remarks

To create a render-target view, call ID3D10Device::CreateRenderTargetView. To bind a render-target view to the pipeline, call ID3D10Device::OMSetRenderTargets.

A rendertarget is a resource that can be written by the output-merger stage at the end of a render pass. Each render-target should also have a corresponding depth-stencil view.

Requirements

Target Platform	Windows
Header	d3d10.h

See also

ID3D10View

Resource Interfaces

ID3D10RenderTargetView::GetDesc method (d3d10.h)

Article • 07/27/20222 minutes to read

Get the properties of a render target view.

Syntax

```
C++

void GetDesc(
  [out] D3D10_RENDER_TARGET_VIEW_DESC *pDesc
);
```

Parameters

`[out] pDesc`

Type: **D3D10_RENDER_TARGET_VIEW_DESC***

Pointer to the description of a render target view (see D3D10_RENDER_TARGET_VIEW_DESC).

Return value

None

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10RenderTargetView Interface

ID3D10Resource interface (d3d10.h)

Article • 07/27/20222 minutes to read

A resource interface provides common actions on all resources.

Inheritance

The **ID3D10Resource** interface inherits from ID3D10DeviceChild. **ID3D10Resource** also has these types of members:

Methods

The **ID3D10Resource** interface has these methods.

ID3D10Resource::GetEvictionPriority Get the eviction priority of a resource. (ID3D10Resource.GetEvictionPriority)
ID3D10Resource::GetType Get the type of the resource. (ID3D10Resource.GetType)
ID3D10Resource::SetEvictionPriority Set the eviction priority of a resource. (ID3D10Resource.SetEvictionPriority)

Remarks

A resource interface cannot be created directly; instead, buffers and textures are created that inherit from a resource interface (see Creating Buffer Resources or Creating Texture Resources).

Requirements

Target Platform	Windows
Header	d3d10.h

See also

ID3D10DeviceChild

Resource Interfaces

ID3D10Resource::GetEvictionPriority method (d3d10.h)

Article • 07/27/20222 minutes to read

Get the eviction priority of a resource.

Syntax

```
C++

UINT GetEvictionPriority();
```

Return value

Type: **UINT**

One of the following values, which specifies the eviction priority for the resource:

- DXGI_RESOURCE_PRIORITY_MINIMUM
- DXGI_RESOURCE_PRIORITY_LOW
- DXGI_RESOURCE_PRIORITY_NORMAL
- DXGI_RESOURCE_PRIORITY_HIGH
- DXGI_RESOURCE_PRIORITY_MAXIMUM

Remarks

This method is a wrapper for GetEvictionPriority and is provided in the ID3D10Resource Interface interface for convenience.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Resource Interface

ID3D10Resource::GetType method (d3d10.h)

Article • 07/27/20222 minutes to read

Get the type of the resource.

Syntax

```
C++

void GetType(
  [out] D3D10_RESOURCE_DIMENSION *rType
);
```

Parameters

`[out] rType`

Type: **D3D10_RESOURCE_DIMENSION***

Pointer to the resource type (see D3D10_RESOURCE_DIMENSION).

Return value

None

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Resource Interface

ID3D10Resource::SetEvictionPriority method (d3d10.h)

Article • 07/27/20222 minutes to read

Set the eviction priority of a resource.

Syntax

```
C++

void SetEvictionPriority(
  [in] UINT EvictionPriority
);
```

Parameters

`[in] EvictionPriority`

Type: **UINT**

Eviction priority for the resource, which is one of the following values:

- DXGI_RESOURCE_PRIORITY_MINIMUM
- DXGI_RESOURCE_PRIORITY_LOW
- DXGI_RESOURCE_PRIORITY_NORMAL
- DXGI_RESOURCE_PRIORITY_HIGH
- DXGI_RESOURCE_PRIORITY_MAXIMUM

Return value

None

Remarks

Resource priorities determine which resource to evict from video memory when the system has run out of video memory. The resource will not be lost; it will be removed from video memory and placed into system memory, or possibly placed onto the hard drive. The resource will be loaded back into video memory when it is required.

A resource that is set to the maximum priority, DXGI_RESOURCE_PRIORITY_MAXIMUM, is only evicted if there is no other way of resolving the incoming memory request. The Windows Display Driver Model (WDDM) tries to split an incoming memory request to its minimum size and evict lower-priority resources before evicting a resource with maximum priority.

Changing the priorities of resources should be done carefully. The wrong eviction priorities could be a detriment to performance rather than an improvement. See QueryResourceResidency for additional information.

This method is a wrapper for SetEvictionPriority and is provided in the ID3D10Resource Interface interface for convenience.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Resource Interface

ID3D10ShaderResourceView interface (d3d10.h)

Article07/27/2022

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, a texture or a sampler.

Inheritance

The **ID3D10ShaderResourceView** interface inherits from ID3D10View. **ID3D10ShaderResourceView** also has these types of members:

Methods

The **ID3D10ShaderResourceView** interface has these methods.

ID3D10ShaderResourceView::GetDesc Get the shader resource view's description. (ID3D10ShaderResourceView.GetDesc)

Remarks

To create a shader-resource view, call ID3D10Device::CreateShaderResourceView.

A shader-resource view is required when binding a resource to a shader stage; the binding occurs by calling ID3D10Device::GSSetShaderResources, ID3D10Device::VSSetShaderResources or ID3D10Device::PSSetShaderResources.

Requirements

Target Platform	Windows
Header	d3d10.h (include D3D10Shader.h)

See also

ID3D10View

Resource Interfaces

ID3D10ShaderResourceView::GetDesc method (d3d10.h)

Article07/27/2022

Get the shader resource view's description.

Syntax

```
C++

void GetDesc(
  [out] D3D10_SHADER_RESOURCE_VIEW_DESC *pDesc
);
```

Parameters

`[out] pDesc`

Type: **D3D10_SHADER_RESOURCE_VIEW_DESC***

A pointer to a D3D10_SHADER_RESOURCE_VIEW_DESC structure to be filled with data about the shader resource view.

Return value

None

Requirements

Target Platform	Windows
Header	d3d10.h (include D3D10Shader.h)

See also

ID3D10ShaderResourceView Interface

ID3D10ShaderResourceView1 interface (d3d10_1.h)

Article07/27/2022

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, a texture or a sampler.

Inheritance

The **ID3D10ShaderResourceView1** interface inherits from ID3D10ShaderResourceView. **ID3D10ShaderResourceView1** also has these types of members:

Methods

The **ID3D10ShaderResourceView1** interface has these methods.

ID3D10ShaderResourceView1::GetDesc1 Get the shader resource view's description. (ID3D10ShaderResourceView1.GetDesc1)

Remarks

To create a shader-resource view, call ID3D10Device1::CreateShaderResourceView1.

A shader-resource view is required when binding a resource to a shader stage; the binding occurs by calling ID3D10Device::GSSetShaderResources, ID3D10Device::VSSetShaderResources or ID3D10Device::PSSetShaderResources.

This method requires Windows Vista Service Pack 1.

Requirements

Target Platform	Windows

Header	d3d10_1.h (include D3D10Shader.h)

See also

ID3D10ShaderResourceView

Resource Interfaces

ID3D10ShaderResourceView1::GetDesc1 method (d3d10_1.h)

Article • 07/27/20222 minutes to read

Get the shader resource view's description.

Syntax

```
C++

void GetDesc1(
  [out] D3D10_SHADER_RESOURCE_VIEW_DESC1 *pDesc
);
```

Parameters

`[out] pDesc`

Type: **D3D10_SHADER_RESOURCE_VIEW_DESC1***

A pointer to a D3D10_SHADER_RESOURCE_VIEW_DESC1 structure to be filled with data about the shader resource view.

Return value

None

Remarks

This method requires Windows Vista Service Pack 1.

Requirements

Target Platform	Windows
Header	d3d10_1.h (include D3D10Shader.h)

See also

ID3D10ShaderResourceView1 Interface

ID3D10Texture1D interface (d3d10.h)

Article • 07/27/20222 minutes to read

A 1D texture interface accesses texel data, which is structured memory.

Inheritance

The **ID3D10Texture1D** interface inherits from ID3D10Resource. **ID3D10Texture1D** also has these types of members:

Methods

The **ID3D10Texture1D** interface has these methods.

ID3D10Texture1D::GetDesc Get the properties of the texture resource. (ID3D10Texture1D.GetDesc)
ID3D10Texture1D::Map Get a pointer to the data contained in a subresource, and deny the GPU access to that subresource.
ID3D10Texture1D::Unmap Invalidate the pointer to a resource that was retrieved by ID3D10Texture1D::Map, and re-enable the GPU's access to that resource.

Remarks

To create an empty 1D texture, call ID3D10Device::CreateTexture1D. For more details on creating and loading textures, see Creating Texture Resources.

Textures cannot be bound directly to the pipeline; instead, a view must be created and bound. Using a view, texture data can be interpreted at run time within certain restrictions. To use the texture as a render target or depth-stencil resource, call ID3D10Device::CreateRenderTargetView, and ID3D10Device::CreateDepthStencilView, respectively. To use the texture as an input to a shader, create a by calling ID3D10Device::CreateShaderResourceView.

Requirements

Target Platform	Windows
Header	d3d10.h

See also

ID3D10Resource

Resource Interfaces

ID3D10Texture1D::GetDesc method (d3d10.h)

Article • 07/27/20222 minutes to read

Get the properties of the texture resource.

Syntax

C++

```
void GetDesc(
  [out] D3D10_TEXTURE1D_DESC *pDesc
);
```

Parameters

`[out] pDesc`

Type: **D3D10_TEXTURE1D_DESC***

Pointer to a resource description (see D3D10_TEXTURE1D_DESC).

Return value

None

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Texture1D Interface

ID3D10Texture1D::Map method (d3d10.h)

Article • 10/13/20212 minutes to read

Get a pointer to the data contained in a subresource, and deny the GPU access to that subresource.

Syntax

```
C++

HRESULT Map(
  [in]  UINT      Subresource,
  [in]  D3D10_MAP MapType,
  [in]  UINT      MapFlags,
  [out] void      **ppData
);
```

Parameters

`[in] Subresource`

Type: **UINT**

Index number of the subresource. See D3D10CalcSubresource for more details.

`[in] MapType`

Type: **D3D10_MAP**

Specifies the CPU's read and write permissions for a resource. For possible values, see D3D10_MAP.

`[in] MapFlags`

Type: **UINT**

Flag that specifies what the CPU should do when the GPU is busy. This flag is optional.

`[out] ppData`

Type: **void****

Pointer to the texture resource data.

Return value

Type: **HRESULT**

If this function succeeds, it returns S_OK. For other restrictions, and a listing of error values that can be returned by any of the **Map** methods, see Remarks.

Remarks

Mapping a texture enables the CPU to directly access the underlying data in the subresource of a texture. For the method to succeed, the texture being mapped must be created with the appropriate flags (see D3D10_CPU_ACCESS_FLAG), and its specified usage (see D3D10_USAGE) must be either D3D10_USAGE_DYNAMIC or D3D10_USAGE_STAGING.

Common failures of **Map** methods are indicated by the following return values:

Item	Description
DXGI_ERROR_WAS_STILL_DRAWING	If *MapFlags* specifies D3D10_MAP_FLAG_DO_NOT_WAIT and the GPU is not yet finished with the resource, **Map** returns DXGI_ERROR_WAS_STILL_DRAWING.
DXGI_ERROR_DEVICE_REMOVED	**Map** returns DXGI_ERROR_DEVICE_REMOVED if *MapType* allows any CPU read access and the video card has been removed.

For more information about the preceding return values, see DXGI_ERROR.

Map has the following restrictions:

- A single subresource cannot be mapped multiple times; in other words, do not call **Map** on a subresource that is already mapped.
- Any subresource that is bound to the pipeline must be unmapped before any render operation (that is, before ID3D10Device::Draw) can be executed.

Applications must cast the void pData pointer to the appropriate type to meaningfully access the underlying subresource data. For example, the following code demonstrates how to read each texel of a 1D subresource. It is assumed that the texture was created

using DXGI_FORMAT_R32G32B32A32_FLOAT and that pData is the pointer to the texture resource data output from a successful call to this method.

```
FLOAT* pTexels = (FLOAT*)pData;
for( UINT col = 0; col < width; col++ )
{
  pTexels[col*4 + 0]; // Alpha
  pTexels[col*4 + 1]; // Blue
  pTexels[col*4 + 2]; // Green
  pTexels[col*4 + 3]; // Red
}
```

Differences between Direct3D 9 and Direct3D 10:
Map in Direct3D 10 is analogous to resource Lock in Direct3D 9.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Texture1D Interface

ID3D10Texture1D::Unmap method (d3d10.h)

Article • 10/13/20212 minutes to read

Invalidate the pointer to a resource that was retrieved by ID3D10Texture1D::Map, and re-enable the GPU's access to that resource.

Syntax

```
C++

void Unmap(
  [in] UINT Subresource
);
```

Parameters

`[in] Subresource`

Type: **UINT**

Subresource to be unmapped. See D3D10CalcSubresource for more details.

Return value

None

Remarks

A subresource must be mapped before Unmap is called.

Differences between Direct3D 9 and Direct3D 10: Unmap in Direct3D 10 is analogous to resource Unlock in Direct3D 9.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Texture1D Interface

ID3D10Texture2D interface (d3d10.h)

Article • 07/27/20222 minutes to read

A 2D texture interface manages texel data, which is structured memory.

Inheritance

The **ID3D10Texture2D** interface inherits from ID3D10Resource. **ID3D10Texture2D** also has these types of members:

Methods

The **ID3D10Texture2D** interface has these methods.

ID3D10Texture2D::GetDesc Get the properties of the texture resource. (ID3D10Texture2D.GetDesc)
ID3D10Texture2D::Map Get a pointer to the data contained in a subresource, and deny GPU access to that subresource. (ID3D10Texture2D.Map)
ID3D10Texture2D::Unmap Invalidate the pointer to the resource that was retrieved by ID3D10Texture2D::Map, and re-enable GPU access to the resource.

Remarks

To create an empty Texture2D resource, call ID3D10Device::CreateTexture2D. For more details on creating and loading textures, see Creating Texture Resources.

Textures cannot be bound directly to the pipeline; instead, a view must be created and bound. Using a view, texture data can be interpreted at run time within certain restrictions. To use the texture as a render target or depth-stencil resource, call ID3D10Device::CreateRenderTargetView, and ID3D10Device::CreateDepthStencilView, respectively. To use the texture as an input to a shader, create a by calling ID3D10Device::CreateShaderResourceView.

Requirements

Target Platform	Windows
Header	d3d10.h

See also

ID3D10Resource

Resource Interfaces

ID3D10Texture2D::GetDesc method (d3d10.h)

Article • 07/27/20222 minutes to read

Get the properties of the texture resource.

Syntax

C++

```
void GetDesc(
  [out] D3D10_TEXTURE2D_DESC *pDesc
);
```

Parameters

`[out] pDesc`

Type: **D3D10_TEXTURE2D_DESC***

Pointer to a resource description (see D3D10_TEXTURE2D_DESC).

Return value

None

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Texture2D Interface

ID3D10Texture2D::Map method (d3d10.h)

Article • 07/27/20222 minutes to read

Get a pointer to the data contained in a subresource, and deny GPU access to that subresource.

Syntax

```
C++

HRESULT Map(
  [in]  UINT                    Subresource,
  [in]  D3D10_MAP               MapType,
  [in]  UINT                    MapFlags,
  [out] D3D10_MAPPED_TEXTURE2D *pMappedTex2D
);
```

Parameters

`[in] Subresource`

Type: **UINT**

Index number of the subresource. See D3D10CalcSubresource for more details.

`[in] MapType`

Type: **D3D10_MAP**

Integer that specifies the CPU's read and write permissions for a resource. For possible values, see D3D10_MAP.

`[in] MapFlags`

Type: **UINT**

Flag that specifies what the CPU should do when the GPU is busy. This flag is optional.

`[out] pMappedTex2D`

Type: **D3D10_MAPPED_TEXTURE2D***

Pointer to a structure (D3D10_MAPPED_TEXTURE2D) that is filled in by the function and contains a pointer to the resource data.

Return value

Type: **HRESULT**

If this function succeeds, it returns S_OK.

All of the Map methods have identical return values and operating restrictions. These are listed in the remarks section of ID3D10Texture1D::Map.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Texture2D Interface

ID3D10Texture2D::Unmap method (d3d10.h)

Article10/13/2021

Invalidate the pointer to the resource that was retrieved by ID3D10Texture2D::Map, and re-enable GPU access to the resource.

Syntax

```
C++

void Unmap(
  [in] UINT Subresource
);
```

Parameters

`[in] Subresource`

Type: **UINT**

Subresource to be unmapped. See D3D10CalcSubresource for more details.

Return value

None

Remarks

A subresource must be mapped before Unmap is called.

Differences between Direct3D 9 and Direct3D 10: Unmap in Direct3D 10 is analogous to resource Unlock in Direct3D 9.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Texture2D Interface

ID3D10Texture3D interface (d3d10.h)

Article • 07/27/20222 minutes to read

A 3D texture interface accesses texel data, which is structured memory.

Inheritance

The **ID3D10Texture3D** interface inherits from ID3D10Resource. **ID3D10Texture3D** also has these types of members:

Methods

The **ID3D10Texture3D** interface has these methods.

ID3D10Texture3D::GetDesc Get the properties of the texture resource. (ID3D10Texture3D.GetDesc)
ID3D10Texture3D::Map Get a pointer to the data contained in a subresource, and deny GPU access to that subresource. (ID3D10Texture3D.Map)
ID3D10Texture3D::Unmap Invalidate the pointer to the resource retrieved by ID3D10Texture3D::Map, and re-enable the GPU's access to the resource.

Remarks

To create an empty Texture3D resource, call ID3D10Device::CreateTexture3D. For more details on creating and loading textures, see Creating Texture Resources.

Textures cannot be bound directly to the pipeline; instead, a view must be created and bound. Using a view, texture data can be interpreted at run time within certain restrictions. To use the texture as a render target or depth-stencil resource, call ID3D10Device::CreateRenderTargetView, and ID3D10Device::CreateDepthStencilView, respectively. To use the texture as an input to a shader, create a by calling ID3D10Device::CreateShaderResourceView.

Requirements

Target Platform	Windows
Header	d3d10.h

See also

ID3D10Resource

Resource Interfaces

ID3D10Texture3D::GetDesc method (d3d10.h)

Article • 07/27/20222 minutes to read

Get the properties of the texture resource.

Syntax

```
C++

void GetDesc(
  [out] D3D10_TEXTURE3D_DESC *pDesc
);
```

Parameters

`[out] pDesc`

Type: **D3D10_TEXTURE3D_DESC***

Pointer to a resource description (see D3D10_TEXTURE3D_DESC).

Return value

None

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Texture3D Interface

ID3D10Texture3D::Map method (d3d10.h)

Article07/27/2022

Get a pointer to the data contained in a subresource, and deny GPU access to that subresource.

Syntax

```
C++

HRESULT Map(
  [in]  UINT                    Subresource,
  [in]  D3D10_MAP               MapType,
  [in]  UINT                    MapFlags,
  [out] D3D10_MAPPED_TEXTURE3D *pMappedTex3D
);
```

Parameters

`[in] Subresource`

Type: **UINT**

Index number of the subresource. See D3D10CalcSubresource for more details.

`[in] MapType`

Type: **D3D10_MAP**

Specifies the CPU's read and write permissions for a resource. For possible values, see D3D10_MAP.

`[in] MapFlags`

Type: **UINT**

Flag that specifies what the CPU should do when the GPU is busy. This flag is optional.

`[out] pMappedTex3D`

Type: **D3D10_MAPPED_TEXTURE3D***

Pointer to a structure (D3D10_MAPPED_TEXTURE3D) that is filled in by the function and contains a pointer to the resource data.

Return value

Type: **HRESULT**

If this function succeeds, it returns S_OK. All of the Map methods have identical return values and operating restrictions. These are listed in the remarks section of ID3D10Texture1D::Map.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Texture3D Interface

ID3D10Texture3D::Unmap method (d3d10.h)

Article10/13/2021

Invalidate the pointer to the resource retrieved by ID3D10Texture3D::Map, and re-enable the GPU's access to the resource.

Syntax

```
C++

void Unmap(
  [in] UINT Subresource
);
```

Parameters

`[in] Subresource`

Type: **UINT**

Subresource to be unmapped. See D3D10CalcSubresource for more details.

Return value

None

Remarks

A subresource must be mapped before Unmap is called.

Differences between Direct3D 9 and Direct3D 10: Unmap() in Direct3D 10 is analogous to resource Unlock() in Direct3D 9.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib

See also

ID3D10Texture3D Interface

ID3D10View interface (d3d10.h)

Article • 07/27/20222 minutes to read

A view interface specifies the parts of a resource the pipeline can access during rendering (see view).

Inheritance

The **ID3D10View** interface inherits from ID3D10DeviceChild. **ID3D10View** also has these types of members:

Methods

The **ID3D10View** interface has these methods.

ID3D10View::GetResource Get the resource that is accessed through this view. (ID3D10View.GetResource)

Remarks

A view interface is the base interface for all views. There are three types of views; a depth-stencil view, a render-target view, and a shader-resource view.

- To create a render-target view, call ID3D10Device::CreateRenderTargetView.
- To create a depth-stencil view, call ID3D10Device::CreateDepthStencilView.
- To create a shader-resource view, call ID3D10Device::CreateShaderResourceView.

All resources must be bound to the pipeline before they can be accessed.

- To bind a render-target view or a depth-stencil view, call ID3D10Device::OMSetRenderTargets.
- To bind a shader-resource view, call ID3D10Device::VSSetShaderResources.

A view can also be used to access a typeless resource.

Requirements

Target Platform	Windows
Header	d3d10.h

See also

ID3D10DeviceChild

Resource Interfaces

ID3D10View::GetResource method (d3d10.h)

Article07/27/2022

Get the resource that is accessed through this view.

Syntax

```
C++

void GetResource(
  [out] ID3D10Resource **ppResource
);
```

Parameters

`[out] ppResource`

Type: **ID3D10Resource****

Address of a pointer to the resource that is accessed through this view. (See ID3D10Resource.)

Return value

None

Remarks

This function increments the reference count of the resource by one, so it is necessary to call Release on the returned pointer when the application is done with it. Destroying (or losing) the returned pointer before Release is called will result in a memory leak.

Requirements

Target Platform	Windows

Header	d3d10.h
Library	D3D10.lib

See also

ID3D10View Interface

Resource Functions (Direct3D 10 Graphics)

Article • 01/06/2021 • 2 minutes to read

This section contains information about the following resource functions:

Functions	Description
D3D10CalcSubresource	Calculate a subresource index.

Related topics

Resource Reference

D3D10CalcSubresource function (d3d10.h)

Article • 10/13/20212 minutes to read

Calculate a subresource index for a texture.

Syntax

```
C++

UINT D3D10CalcSubresource(
  [in] UINT MipSlice,
  [in] UINT ArraySlice,
  [in] UINT MipLevels
);
```

Parameters

`[in] MipSlice`

Type: **UINT**

A zero-based index into an array of subtextures; 0 indicates the first, most detailed subtexture (or mipmap level).

`[in] ArraySlice`

Type: **UINT**

The zero-based index of the first texture to use (in an array of textures).

`[in] MipLevels`

Type: **UINT**

Number of mipmap levels (or subtextures) to use.

Return value

Type: **inline UINT**

The index which equals *MipSlice* + (*ArraySlice* * *MipLevels*).

Remarks

A buffer is an unstructured resource and is therefore defined as containing a single subresource. APIs that take buffers do not need a subresource index. A texture on the other hand is highly structured. Each texture object may contain one or more subresources depending on the size of the array and the number of mipmap levels.

Requirements

Target Platform	Windows
Header	d3d10.h
Library	D3D10.lib
DLL	D3D10.dll

See also

Core Functions

Resource Functions

Resource Structures (Direct3D 10 Graphics)

Article • 01/06/2021 • 2 minutes to read

Structures used to create and use resources.

Structures	Description
D3D10_BUFFER_DESC	Description of a buffer resource.
D3D10_BUFFER_RTV	Describes the elements in a buffer to be used in a render-target view.
D3D10_BUFFER_SRV	Describes the elements in a buffer resource to use in a shader-resource view.
D3D10_DEPTH_STENCIL_VIEW_DESC	Describes a depth-stencil view.
D3D10_MAPPED_TEXTURE2D	Provides access to subresource data in a 2D texture.
D3D10_MAPPED_TEXTURE3D	Provides access to subresource data in a 3D texture.
D3D10_RENDER_TARGET_VIEW_DESC	Describes a render-target view.
D3D10_SUBRESOURCE_DATA	Specifies data for initializing a resource.
D3D10_TEX1D_ARRAY_DSV	Specifies which mipmap level and textures in a 1D-texture array to use in a depth-stencil view.
D3D10_TEX1D_ARRAY_RTV	Specifies the subresource(s) from an array of 1D textures to use in a render-target view.
D3D10_TEX1D_ARRAY_SRV	Specifies the mipmap levels and textures in a 1D-texture array to use in a shader-resource view.
D3D10_TEX1D_DSV	Specifies the subresource(s) from a 1D texture to use in a depth-stencil view.
D3D10_TEX1D_RTV	Specifies the subresource(s) from a 1D texture to use in a render-target view.
D3D10_TEX1D_SRV	Specifies the subresource(s) from a 1D texture to use in a shader-resource view.
D3D10_TEX2D_ARRAY_DSV	Specifies which mipmap level and which textures in a 2D-texture array to use in a depth-stencil view.
D3D10_TEX2D_ARRAY_RTV	Specifies which mipmap level and which textures in a 2D-texture array to use in a render-target view.

Structures	Description
D3D10_TEX2D_ARRAY_SRV	Specifies the subresource(s) from an array of 2D textures to use in a shader-resource view.
D3D10_TEX2D_DSV	Specifies the subresource(s) from a 2D texture to use in a depth-stencil view.
D3D10_TEX2D_RTV	Specifies the subresource(s) from a 2D texture to use in a render-target view.
D3D10_TEX2D_SRV	Specifies the subresource(s) from a 2D texture to use in a shader-resource view.
D3D10_TEX2DMS_ARRAY_DSV	Specifies the subresource(s) from an array of 2D textures to use in a depth-stencil view.
D3D10_TEX2DMS_ARRAY_RTV	Specifies the subresource(s) from an array of 2D textures to use in a render-target view.
D3D10_TEX2DMS_ARRAY_SRV	Specifies the subresource(s) from an array of 2D textures to use in a shader-resource view.
D3D10_TEX2DMS_DSV	Specifies the subresource(s) from a multisampled 2D texture to use in a depth-stencil view.
D3D10_TEX2DMS_RTV	Specifies the subresource(s) from a multisampled 2D texture to use in a render-target view.
D3D10_TEX2DMS_SRV	Specifies the subresource(s) from a multisampled 2D texture to use in a shader-resource view.
D3D10_TEX3D_RTV	Specifies the subresource(s) from a 3D texture to use in a render-target view.
D3D10_TEX3D_SRV	Specifies the subresource(s) from a 3D texture to use in a shader-resource view.
D3D10_TEXCUBE_ARRAY_SRV1	Specifies the subresource(s) from a cube texture to use in a shader-resource view.
D3D10_TEXCUBE_SRV	Specifies the subresource(s) from a cube texture to use in a shader-resource view.
D3D10_TEXTURE1D_DESC	Describes a 1D texture resource.
D3D10_TEXTURE2D_DESC	Describes a 2D texture resource.
D3D10_TEXTURE3D_DESC	Describes a 3D texture resource.

Related topics

Resource Reference

CD3D10_BUFFER_DESC structure (d3d10.h)

Article • 02/16/20232 minutes to read

Describes a buffer resource.

Syntax

```
C++

struct CD3D10_BUFFER_DESC : D3D10_BUFFER_DESC {
  void CD3D10_BUFFER_DESC();
  void CD3D10_BUFFER_DESC(
    const D3D10_BUFFER_DESC & o
  );
  void CD3D10_BUFFER_DESC(
    UINT        byteWidth,
    UINT        bindFlags,
    D3D10_USAGE usage,
    UINT        cpuaccessFlags,
    UINT        miscFlags
  );
  void ~CD3D10_BUFFER_DESC();
};
```

Inheritance

The **CD3D10_BUFFER_DESC** structure implements D3D10_BUFFER_DESC.

Members

`void CD3D10_BUFFER_DESC()`

`void CD3D10_BUFFER_DESC( const D3D10_BUFFER_DESC & o)`

`void CD3D10_BUFFER_DESC( UINT byteWidth, UINT bindFlags, D3D10_USAGE usage, UINT cpuaccessFlags, UINT miscFlags)`

`void ~CD3D10_BUFFER_DESC()`

Remarks

`byteWidth` Type: **UINT** Size of the buffer in bytes.

`bindFlags` Type: **UINT** Identify how the buffer will be bound to the pipeline. Applications can logically OR flags together (see D3D10_BIND_FLAG) to indicate that the buffer can be accessed in different ways.

`usage` Type: **D3D10_USAGE** Identify how the buffer is expected to be read from and written to. Frequency of update is a key factor. The most common value is typically D3D10_USAGE_DEFAULT; see D3D10_USAGE for all possible values.

`cpuAccessFlags` Type: **UINT** CPU access flags (see D3D10_CPU_ACCESS_FLAG) or 0 if no CPU access is necessary. Applications can logically OR flags together.

`miscFlags` Type: **UINT** Miscellaneous flags (see D3D10_RESOURCE_MISC_FLAG) or 0 if unused. Applications can logically OR flags together.

This structure is used by ID3D10Device::CreateBuffer to create buffer resources.

In addition to this structure, there is also a derived structure in D3D10.h (CD3D10_BUFFER_DESC) which behaves like an inherited class to help create a buffer description.

Requirements

Header	d3d10.h

See also

Resource Structures

D3D10_BUFFER_RTV structure (d3d10.h)

Article • 04/02/20212 minutes to read

Specifies the elements from a buffer resource to use in a render-target view.

Syntax

C++

```
typedef struct D3D10_BUFFER_RTV {
  union {
    UINT FirstElement;
    UINT ElementOffset;
  };
  union {
    UINT NumElements;
    UINT ElementWidth;
  };
} D3D10_BUFFER_RTV;
```

Members

`FirstElement`

Type: **UINT**

Number of bytes between the beginning of the buffer and the first element to access.

`ElementOffset`

Type: **UINT**

The offset of the first element in the view to access, relative to element 0.

`NumElements`

Type: **UINT**

The total number of elements in the view.

`ElementWidth`

Type: **UINT**

The width of each element (in bytes). This can be determined from the format stored in the render-target-view description.

Remarks

A render-target view is a member of a render-target-view description (see D3D10_RENDER_TARGET_VIEW_DESC). Create a render-target view by calling ID3D10Device::CreateRenderTargetView.

Requirements

Header	d3d10.h

See also

Resource Structures

D3D10_BUFFER_SRV structure (d3d10.h)

Article • 07/27/20222 minutes to read

Specifies the elements in a buffer resource to use in a shader-resource view.

Syntax

C++

```
typedef struct D3D10_BUFFER_SRV {
  union {
    UINT FirstElement;
    UINT ElementOffset;
  };
  union {
    UINT NumElements;
    UINT ElementWidth;
  };
} D3D10_BUFFER_SRV;
```

Members

`FirstElement`

Type: **UINT**

Number of bytes between the beginning of the buffer and the first element to access.

`ElementOffset`

Type: **UINT**

The offset of the first element in the view to access, relative to element 0.

`NumElements`

Type: **UINT**

The total number of elements in the view.

`ElementWidth`

Type: **UINT**

The width of each element (in bytes). This can be determined from the format stored in the shader-resource-view description.

Remarks

The **D3D10_BUFFER_SRV** structure is a member of the D3D10_SHADER_RESOURCE_VIEW_DESC structure, which represents a shader-resource view description. You can create a shader-resource view by calling the ID3D10Device::CreateShaderResourceView method.

Requirements

Header	d3d10.h

See also

Resource Structures

D3D10_DEPTH_STENCIL_VIEW_DESC structure (d3d10.h)

Article • 04/02/20212 minutes to read

Specifies the subresource(s) from a texture that are accessible using a depth-stencil view.

Syntax

C++

```
typedef struct D3D10_DEPTH_STENCIL_VIEW_DESC {
  DXGI_FORMAT         Format;
  D3D10_DSV_DIMENSION ViewDimension;
  union {
    D3D10_TEX1D_DSV         Texture1D;
    D3D10_TEX1D_ARRAY_DSV   Texture1DArray;
    D3D10_TEX2D_DSV         Texture2D;
    D3D10_TEX2D_ARRAY_DSV   Texture2DArray;
    D3D10_TEX2DMS_DSV       Texture2DMS;
    D3D10_TEX2DMS_ARRAY_DSV Texture2DMSArray;
  };
} D3D10_DEPTH_STENCIL_VIEW_DESC;
```

Members

`Format`

Type: **DXGI_FORMAT**

Resource data format (see DXGI_FORMAT). See remarks for allowable formats.

`ViewDimension`

Type: **D3D10_DSV_DIMENSION**

Type of resource (see D3D10_DSV_DIMENSION). Specifies how a depth-stencil resource will be accessed; the value is stored in the union in this structure.

`Texture1D`

Type: **D3D10_TEX1D_DSV**

Specifies a 1D texture subresource (see D3D10_TEX1D_DSV).

`Texture1DArray`

Type: **D3D10_TEX1D_ARRAY_DSV**

Specifies an array of 1D texture subresources (see D3D10_TEX1D_ARRAY_DSV).

`Texture2D`

Type: **D3D10_TEX2D_DSV**

Specifies a 2D texture subresource (see D3D10_TEX2D_DSV).

`Texture2DArray`

Type: **D3D10_TEX2D_ARRAY_DSV**

Specifies an array of 2D texture subresources (see D3D10_TEX2D_ARRAY_DSV).

`Texture2DMS`

Type: **D3D10_TEX2DMS_DSV**

Specifies a multisampled 2D texture contains a single subresource (see D3D10_TEX2DMS_DSV).

`Texture2DMSArray`

Type: **D3D10_TEX2DMS_ARRAY_DSV**

Specifies a multisampled 2D texture contains a single subresource per texture (see D3D10_TEX2DMS_ARRAY_DSV).

Remarks

These are valid formats for a depth-stencil view:

- DXGI_FORMAT_D16_UNORM
- DXGI_FORMAT_D24_UNORM_S8_UINT
- DXGI_FORMAT_D32_FLOAT
- DXGI_FORMAT_D32_FLOAT_S8X24_UINT
- DXGI_FORMAT_UNKNOWN

A depth-stencil view cannot use a typeless format. If the format chosen is DXGI_FORMAT_UNKNOWN, then the format of the parent resource is used.

A depth-stencil-view description is needed when calling ID3D10Device::CreateDepthStencilView.

Requirements

Header	d3d10.h

See also

Resource Structures

D3D10_MAPPED_TEXTURE2D structure (d3d10.h)

Article04/02/2021

Provides access to subresource data in a 2D texture.

Syntax

C++

```
typedef struct D3D10_MAPPED_TEXTURE2D {
  void *pData;
  UINT RowPitch;
} D3D10_MAPPED_TEXTURE2D;
```

Members

`pData`

Type: **void***

Pointer to the data.

`RowPitch`

Type: **UINT**

The pitch, or width, or physical size (in bytes), of one row of an uncompressed texture. A block-compressed texture is encoded in 4x4 blocks (see virtual size vs physical size) ; therefore, **RowPitch** is the number of bytes in a block of 4x4 texels.

Remarks

This structure is used in a call to Map.

To illustrate the row pitch, assume an uncompressed 2D texture with mipmap levels, as shown in the following illustration.

Visualize the top-level texture drawn in a single plane like the following illustration.

However, the actual layout of each element in memory looks more like the following illustration.

For this example, the row pitch encompasses 5 elements (one row), whose size would be five times the number of bytes per element.

Use row pitch to advance a pointer between rows within a single 2D texture plane.

Example

To access data in, say, the third mipmap level, you must cast the **pData** pointer as demonstrated in the following example for a floating-point texture.

```
D3D10_MAPPED_TEXTURE2D mappedTexture;
if( SUCCEEDED( pTexture->Map( D3D10CalcSubresource(2, 0, 3),
D3D10_MAP_WRITE_DISCARD, 0, &mappedTexture )))
{
    D3D10_TEXTURE2D_DESC desc;
    pTexture->GetDesc( &desc );

    // Compute the width and height of the third mipmap level
    const UINT WIDTH = desc.Width >> 2;
    const UINT HEIGHT = desc.Height >> 2;

    FLOAT* pTexels = (FLOAT*)mappedTexture.pData;
    for( UINT row = 0; row < HEIGHT; row++ )
    {
      UINT rowStart = row * mappedTexture.RowPitch/4;
      for( UINT col = 0; col < WIDTH; col++ )
      {
        pTexels[rowStart + col*4 + 0]; // Red
```

```
        pTexels[rowStart + col*4 + 1]; // Green
        pTexels[rowStart + col*4 + 2]; // Blue
        pTexels[rowStart + col*4 + 3]; // Alpha
      }
    }

    pTexture->Unmap(D3D10CalcSubresource(2, 0, 3));
}
```

Requirements

Header	d3d10.h

See also

Resource Structures

D3D10_MAPPED_TEXTURE3D structure (d3d10.h)

Article04/02/2021

Provides access to subresource data in a 3D texture.

Syntax

C++

```
typedef struct D3D10_MAPPED_TEXTURE3D {
  void *pData;
  UINT RowPitch;
  UINT DepthPitch;
} D3D10_MAPPED_TEXTURE3D;
```

Members

`pData`

Type: **void***

Pointer to the data.

`RowPitch`

Type: **UINT**

The pitch, or width, or physical size (in bytes) of one row of an uncompressed texture. Since a block-compressed texture is encoded in 4x4 blocks, the **RowPitch** for a compressed texture is the number of bytes in a block of 4x4 texels. See virtual size vs physical size for more information on block compression.

`DepthPitch`

Type: **UINT**

The pitch or number of bytes in all rows for a single depth.

Remarks

This structure is used to access subresource data when calling ID3D10Texture3D::Map. To access data, you must cast the **pData** pointer; see D3D10_MAPPED_TEXTURE2D for an example.

To illustrate pitch for an uncompressed texture, assume a 3D texture with mipmap levels, as shown in the following illustration.

It is easiest to consider the top-level texture only, as shown in the following illustration.

And then visualize the top-level texture redrawn as a series of 2D textures, each one having a different depth value. This yields several texture planes, as shown in the following illustration.

However, the actual layout of all the elements from all the texture planes looks more like the following illustration.

Use row pitch to advance a pointer between rows within a single 2D texture plane; use depth pitch to advance a pointer between 2D texture planes.

Requirements

Header	d3d10.h

See also

Resource Structures

D3D10_RENDER_TARGET_VIEW_DESC structure (d3d10.h)

Article • 04/02/20212 minutes to read

Specifies the subresource(s) from a resource that are accessible using a render-target view.

Syntax

C++

```
typedef struct D3D10_RENDER_TARGET_VIEW_DESC {
  DXGI_FORMAT         Format;
  D3D10_RTV_DIMENSION ViewDimension;
  union {
    D3D10_BUFFER_RTV        Buffer;
    D3D10_TEX1D_RTV         Texture1D;
    D3D10_TEX1D_ARRAY_RTV   Texture1DArray;
    D3D10_TEX2D_RTV         Texture2D;
    D3D10_TEX2D_ARRAY_RTV   Texture2DArray;
    D3D10_TEX2DMS_RTV       Texture2DMS;
    D3D10_TEX2DMS_ARRAY_RTV Texture2DMSArray;
    D3D10_TEX3D_RTV         Texture3D;
  };
} D3D10_RENDER_TARGET_VIEW_DESC;
```

Members

`Format`

Type: **DXGI_FORMAT**

The data format (see DXGI_FORMAT).

`ViewDimension`

Type: **D3D10_RTV_DIMENSION**

The resource type (see D3D10_RTV_DIMENSION), which specifies how the render-target resource will be accessed.

`Buffer`

Type: **D3D10_BUFFER_RTV**

Specifies which buffer elements can be accessed (see D3D10_BUFFER_RTV).

`Texture1D`

Type: **D3D10_TEX1D_RTV**

Specifies the subresources in a 1D texture that can be accessed (see D3D10_TEX1D_RTV).

`Texture1DArray`

Type: **D3D10_TEX1D_ARRAY_RTV**

Specifies the subresources in a 1D texture array that can be accessed (see D3D10_TEX1D_ARRAY_RTV).

`Texture2D`

Type: **D3D10_TEX2D_RTV**

Specifies the subresources in a 2D texture that can be accessed (see D3D10_TEX2D_RTV).

`Texture2DArray`

Type: **D3D10_TEX2D_ARRAY_RTV**

Specifies the subresources in a 2D texture array that can be accessed (see D3D10_TEX2D_ARRAY_RTV).

`Texture2DMS`

Type: **D3D10_TEX2DMS_RTV**

Specifies a single subresource because a multisampled 2D texture only contains one subresource (see D3D10_TEX2DMS_RTV).

`Texture2DMSArray`

Type: **D3D10_TEX2DMS_ARRAY_RTV**

Specifies the subresources in a multisampled 2D texture array that can be accessed (see D3D10_TEX2DMS_ARRAY_RTV).

`Texture3D`

Type: **D3D10_TEX3D_RTV**

Specifies subresources in a 3D texture that can be accessed (see D3D10_TEX3D_RTV).

Remarks

A render-target-view description is passed into ID3D10Device::CreateRenderTargetView to create a render target.

A render-target-view cannot use the following formats:

- Any typeless format.
- DXGI_FORMAT if the view will be used to bind a buffer (vertex, index, constant, or stream-output).

If the format is set to DXGI_FORMAT, then the format of the resource that the view binds to the pipeline will be used.

Requirements

Header	d3d10.h

See also

Resource Structures

D3D10_SUBRESOURCE_DATA structure (d3d10.h)

Article • 07/27/20222 minutes to read

Specifies data for initializing a subresource.

Syntax

C++

```
typedef struct D3D10_SUBRESOURCE_DATA {
  const void *pSysMem;
  UINT       SysMemPitch;
  UINT       SysMemSlicePitch;
} D3D10_SUBRESOURCE_DATA;
```

Members

`pSysMem`

Type: **const void***

Pointer to the initialization data.

`SysMemPitch`

Type: **UINT**

The distance (in bytes) from the beginning of one line of a texture to the next line. System-memory pitch is used only for 2D and 3D texture data as it is has no meaning for the other resource types.

`SysMemSlicePitch`

Type: **UINT**

The distance (in bytes) from the beginning of one depth level to the next. System-memory-slice pitch is only used for 3D texture data as it has no meaning for the other resource types.

Remarks

This structure is used in calls to create buffers (ID3D10Device::CreateBuffer) and textures (ID3D10Device::CreateTexture1D, ID3D10Device::CreateTexture2D, and ID3D10Device::CreateTexture3D). If the resource being created does not require a system-memory pitch or a system-memory-pitch slice, then you are free to use those members to pass size information which may help you when debugging a problem creating a resource.

Note that an application should not rely on **SysMemPitch** being exactly equal to the number of texels in a line times the size of a texel. In some cases **SysMemPitch** will include padding to skip past additional data in a line. This could be padding for alignment or the texture could be a subsection of a larger texture. For example the D3D10_SUBRESOURCE_DATA structure could represent a 32 by 32 subsection of a 128 by 128 texture.
The value for **SysMemSlicePitch** will reflect any padding included in **SysMemPitch**.

Requirements

Header	d3d10.h

See also

Resource Structures

D3D10_TEX1D_ARRAY_DSV structure (d3d10.h)

Article • 04/02/20212 minutes to read

Specifies the subresource(s) from an array of 1D textures to use in a depth-stencil view.

Syntax

C++

```
typedef struct D3D10_TEX1D_ARRAY_DSV {
  UINT MipSlice;
  UINT FirstArraySlice;
  UINT ArraySize;
} D3D10_TEX1D_ARRAY_DSV;
```

Members

`MipSlice`

Type: **UINT**

The index of the first mipmap level to use (see mip slice).

`FirstArraySlice`

Type: **UINT**

The index of the first texture to use in an array of textures (see array slice)

`ArraySize`

Type: **UINT**

Number of textures to use.

Remarks

This structure is one member of a depth-stencil-view description (see D3D10_DEPTH_STENCIL_VIEW_DESC).

Requirements

Header	d3d10.h

See also

Resource Structures

D3D10_TEX1D_ARRAY_RTV structure (d3d10.h)

Article • 04/02/20212 minutes to read

Specifies the subresource(s) from an array of 1D textures to use in a render-target view.

Syntax

C++

```
typedef struct D3D10_TEX1D_ARRAY_RTV {
  UINT MipSlice;
  UINT FirstArraySlice;
  UINT ArraySize;
} D3D10_TEX1D_ARRAY_RTV;
```

Members

`MipSlice`

Type: **UINT**

The index of the mipmap level to use (see mip slice).

`FirstArraySlice`

Type: **UINT**

The index of the first texture to use in an array of textures (see array slice)

`ArraySize`

Type: **UINT**

Number of textures to use.

Remarks

This structure is one member of a render-target-view description (see D3D10_RENDER_TARGET_VIEW_DESC).

Requirements

Header	d3d10.h

See also

Resource Structures

D3D10_TEX1D_ARRAY_SRV structure (d3d10.h)

Article04/02/2021

Specifies the subresource(s) from an array of 1D textures to use in a shader-resource view.

Syntax

```
C++

typedef struct D3D10_TEX1D_ARRAY_SRV {
  UINT MostDetailedMip;
  UINT MipLevels;
  UINT FirstArraySlice;
  UINT ArraySize;
} D3D10_TEX1D_ARRAY_SRV;
```

Members

`MostDetailedMip`

Type: **UINT**

Index of the most detailed mipmap level to use; this number is between 0 and **MipLevels** -1.

`MipLevels`

Type: **UINT**

Number of subtextures to access.

`FirstArraySlice`

Type: **UINT**

The index of the first texture to use in an array of textures (see array slice)

`ArraySize`

Type: **UINT**

Number of textures in the array.

Remarks

This structure is one member of a shader-resource-view description (see D3D10_SHADER_RESOURCE_VIEW_DESC).

Requirements

Header	d3d10.h

See also

Resource Structures

D3D10_TEX1D_DSV structure (d3d10.h)

Article07/27/2022

Specifies the subresource from a 1D texture that is accessible to a depth-stencil view.

Syntax

```
C++

typedef struct D3D10_TEX1D_DSV {
  UINT MipSlice;
} D3D10_TEX1D_DSV;
```

Members

`MipSlice`

Type: **UINT**

The index of the first mipmap level to use (see mip slice).

Remarks

This structure is one member of a depth-stencil-view description (see D3D10_DEPTH_STENCIL_VIEW_DESC).

Requirements

Header	d3d10.h

See also

Resource Structures

D3D10_TEX1D_RTV structure (d3d10.h)

Article • 07/27/20222 minutes to read

Specifies the subresource from a 1D texture to use in a render-target view.

Syntax

```C++
typedef struct D3D10_TEX1D_RTV {
  UINT MipSlice;
} D3D10_TEX1D_RTV;
```

Members

`MipSlice`

Type: **UINT**

The index of the mipmap level to use (see mip slice).

Remarks

This structure is one member of a render-target-view description (see D3D10_RENDER_TARGET_VIEW_DESC).

Requirements

Header	d3d10.h

See also

Resource Structures

D3D10_TEX1D_SRV structure (d3d10.h)

Article • 07/27/20222 minutes to read

Specifies the subresource from a 1D texture to use in a shader-resource view.

Syntax

```
C++

typedef struct D3D10_TEX1D_SRV {
  UINT MostDetailedMip;
  UINT MipLevels;
} D3D10_TEX1D_SRV;
```

Members

`MostDetailedMip`

Type: **UINT**

Index of the most detailed mipmap level to use; this number is between 0 and **MipLevels** -1.

`MipLevels`

Type: **UINT**

Number of mipmap levels to use.

Remarks

This structure is one member of a shader-resource-view description (see D3D10_SHADER_RESOURCE_VIEW_DESC).

Requirements

Header	d3d10.h

See also

Resource Structures

D3D10_TEX2D_ARRAY_DSV structure (d3d10.h)

Article04/02/2021

Specifies the subresource(s) from an array 2D textures that are accessible to a depth-stencil view.

Syntax

```C++
typedef struct D3D10_TEX2D_ARRAY_DSV {
  UINT MipSlice;
  UINT FirstArraySlice;
  UINT ArraySize;
} D3D10_TEX2D_ARRAY_DSV;
```

Members

`MipSlice`

Type: **UINT**

The index of the first mipmap level to use (see mip slice).

`FirstArraySlice`

Type: **UINT**

The index of the first texture to use in an array of textures (see array slice)

`ArraySize`

Type: **UINT**

Number of textures to use.

Remarks

This structure is one member of a depth-stencil-view description (see D3D10_DEPTH_STENCIL_VIEW_DESC).

Requirements

Header	d3d10.h

See also

Resource Structures

D3D10_TEX2D_ARRAY_RTV structure (d3d10.h)

Article • 04/02/20212 minutes to read

Specifies the subresource(s) from an array of 2D textures to use in a render-target view.

Syntax

C++

```
typedef struct D3D10_TEX2D_ARRAY_RTV {
  UINT MipSlice;
  UINT FirstArraySlice;
  UINT ArraySize;
} D3D10_TEX2D_ARRAY_RTV;
```

Members

`MipSlice`

Type: **UINT**

The index of the mipmap level to use (see mip slice).

`FirstArraySlice`

Type: **UINT**

The index of the first texture to use in an array of textures (see array slice)

`ArraySize`

Type: **UINT**

Number of textures in the array to use in the render target view, starting from **FirstArraySlice**.

Remarks

This structure is one member of a render-target-view description (see D3D10_RENDER_TARGET_VIEW_DESC).

Requirements

Header	d3d10.h

See also

Resource Structures

D3D10_TEX2D_ARRAY_SRV structure (d3d10.h)

Article • 04/02/20212 minutes to read

Specifies the subresource(s) from an array of 2D textures to use in a shader-resource view.

Syntax

```C++
typedef struct D3D10_TEX2D_ARRAY_SRV {
  UINT MostDetailedMip;
  UINT MipLevels;
  UINT FirstArraySlice;
  UINT ArraySize;
} D3D10_TEX2D_ARRAY_SRV;
```

Members

`MostDetailedMip`

Type: **UINT**

Index of the most detailed mipmap level to use; this number is between 0 and **MipLevels** -1.

`MipLevels`

Type: **UINT**

Number of subtextures to access.

`FirstArraySlice`

Type: **UINT**

The index of the first texture to use in an array of textures (see array slice)

`ArraySize`

Type: **UINT**

Number of textures in the array.

Remarks

This structure is one member of a shader-resource-view description (see D3D10_SHADER_RESOURCE_VIEW_DESC).

Requirements

Header	d3d10.h

See also

Resource Structures

D3D10_TEX2D_DSV structure (d3d10.h)

Article • 07/27/20222 minutes to read

Specifies the subresource from a 2D texture that is accessible to a depth-stencil view.

Syntax

C++

```
typedef struct D3D10_TEX2D_DSV {
  UINT MipSlice;
} D3D10_TEX2D_DSV;
```

Members

`MipSlice`

Type: **UINT**

The index of the first mipmap level to use (see mip slice).

Remarks

This structure is one member of a depth-stencil-view description (see D3D10_DEPTH_STENCIL_VIEW_DESC).

Requirements

Header	d3d10.h

See also

Resource Structures

D3D10_TEX2D_RTV structure (d3d10.h)

Article07/27/2022

Specifies the subresource from a 2D texture to use in a render-target view.

Syntax

```C++
typedef struct D3D10_TEX2D_RTV {
  UINT MipSlice;
} D3D10_TEX2D_RTV;
```

Members

`MipSlice`

Type: **UINT**

The index of the mipmap level to use (see mip slice).

Remarks

This structure is one member of a render-target-view description (see D3D10_RENDER_TARGET_VIEW_DESC).

Requirements

Header	d3d10.h

See also

Resource Structures

D3D10_TEX2D_SRV structure (d3d10.h)

Article • 07/27/20222 minutes to read

Specifies the subresource from a 2D texture to use in a shader-resource view.

Syntax

C++

```
typedef struct D3D10_TEX2D_SRV {
  UINT MostDetailedMip;
  UINT MipLevels;
} D3D10_TEX2D_SRV;
```

Members

`MostDetailedMip`

Type: **UINT**

Index of the most detailed mipmap level to use; this number is between 0 and **MipLevels** -1.

`MipLevels`

Type: **UINT**

Number of mipmap levels to use.

Remarks

This structure is one member of a shader-resource-view description (see D3D10_SHADER_RESOURCE_VIEW_DESC).

Requirements

Header	d3d10.h

See also

Resource Structures

D3D10_TEX2DMS_ARRAY_DSV structure (d3d10.h)

Article • 04/02/20212 minutes to read

Specifies the subresource(s) from an array of multisampled 2D textures for a depth-stencil view.

Syntax

```
C++

typedef struct D3D10_TEX2DMS_ARRAY_DSV {
  UINT FirstArraySlice;
  UINT ArraySize;
} D3D10_TEX2DMS_ARRAY_DSV;
```

Members

`FirstArraySlice`

Type: **UINT**

The index of the first texture to use in an array of textures (see array slice)

`ArraySize`

Type: **UINT**

Number of textures to use.

Remarks

This structure is one member of a depth-stencil-view description (see D3D10_DEPTH_STENCIL_VIEW_DESC).

Requirements

Header	d3d10.h

See also

Resource Structures

D3D10_TEX2DMS_ARRAY_RTV structure (d3d10.h)

Article • 04/02/20212 minutes to read

Specifies the subresource(s) from a an array of multisampled 2D textures to use in a render-target view.

Syntax

```
C++

typedef struct D3D10_TEX2DMS_ARRAY_RTV {
  UINT FirstArraySlice;
  UINT ArraySize;
} D3D10_TEX2DMS_ARRAY_RTV;
```

Members

`FirstArraySlice`

Type: **UINT**

The index of the first texture to use in an array of textures (see array slice)

`ArraySize`

Type: **UINT**

Number of textures to use.

Remarks

This structure is one member of a render-target-view description (see D3D10_RENDER_TARGET_VIEW_DESC).

Requirements

Header	d3d10.h

See also

Resource Structures

D3D10_TEX2DMS_ARRAY_SRV structure (d3d10.h)

Article • 04/02/20212 minutes to read

Specifies the subresource(s) from an array of multisampled 2D textures to use in a shader-resource view.

Syntax

```C++
typedef struct D3D10_TEX2DMS_ARRAY_SRV {
  UINT FirstArraySlice;
  UINT ArraySize;
} D3D10_TEX2DMS_ARRAY_SRV;
```

Members

`FirstArraySlice`

Type: **UINT**

The index of the first texture to use in an array of textures (see array slice)

`ArraySize`

Type: **UINT**

Number of textures to use.

Remarks

This structure is one member of a shader-resource-view description (see D3D10_SHADER_RESOURCE_VIEW_DESC).

Requirements

Header	d3d10.h

See also

Resource Structures

D3D10_TEX2DMS_DSV structure (d3d10.h)

Article • 07/27/20222 minutes to read

Specifies the subresource from a multisampled 2D texture that is accessible to a depth-stencil view.

Syntax

C++

```
typedef struct D3D10_TEX2DMS_DSV {
  UINT UnusedField_NothingToDefine;
} D3D10_TEX2DMS_DSV;
```

Members

`UnusedField_NothingToDefine`

Type: **UINT**

Unused.

Remarks

Since a multisampled 2D texture contains a single subtexture, there is nothing to specify; this unused member is included so that this structure will compile in C.

Requirements

Header	d3d10.h

See also

Resource Structures

D3D10_TEX2DMS_RTV structure (d3d10.h)

Article07/27/2022

Specifies the subresource from a multisampled 2D texture to use in a render-target view.

Syntax

```C++
typedef struct D3D10_TEX2DMS_RTV {
  UINT UnusedField_NothingToDefine;
} D3D10_TEX2DMS_RTV;
```

Members

`UnusedField_NothingToDefine`

Type: **UINT**

Integer of any value. See remarks.

Remarks

Since a multisampled 2D texture contains a single subresource, there is actually nothing to specify in **D3D10_TEX2DMS_RTV**. Consequently, **UnusedField_NothingToDefine** is included so that this structure will compile in C.

Requirements

Header	d3d10.h

See also

Resource Structures

D3D10_TEX2DMS_SRV structure (d3d10.h)

Article • 04/02/20212 minutes to read

Specifies the subresource(s) from a multisampled 2D texture to use in a shader-resource view.

Syntax

```
C++

typedef struct D3D10_TEX2DMS_SRV {
  UINT UnusedField_NothingToDefine;
} D3D10_TEX2DMS_SRV;
```

Members

`UnusedField_NothingToDefine`

Type: **UINT**

Integer of any value. See remarks.

Remarks

Since a multisampled 2D texture contains a single subresource, there is actually nothing to specify in D3D10_TEX2DMS_RTV. Consequently, **UnusedField_NothingToDefine** is included so that this structure will compile in C.

Requirements

Header	d3d10.h

See also

Resource Structures

D3D10_TEX3D_RTV structure (d3d10.h)

Article • 04/02/20212 minutes to read

Specifies the subresource(s) from a 3D texture to use in a render-target view.

Syntax

```
C++

typedef struct D3D10_TEX3D_RTV {
  UINT MipSlice;
  UINT FirstWSlice;
  UINT WSize;
} D3D10_TEX3D_RTV;
```

Members

`MipSlice`

Type: **UINT**

The index of the mipmap level to use (see mip slice).

`FirstWSlice`

Type: **UINT**

First depth level to use.

`WSize`

Type: **UINT**

Number of depth levels to use in the render-target view, starting from **FirstWSlice**. A value of -1 indicates all of the slices along the w axis, starting from **FirstWSlice**.

Remarks

This structure is one member of a render target view. See D3D10_RENDER_TARGET_VIEW_DESC.

Requirements

Header	d3d10.h

See also

Resource Structures

D3D10_TEX3D_SRV structure (d3d10.h)

Article • 04/02/20212 minutes to read

Specifies the subresource(s) from a 3D texture to use in a shader-resource view.

Syntax

```
C++

typedef struct D3D10_TEX3D_SRV {
  UINT MostDetailedMip;
  UINT MipLevels;
} D3D10_TEX3D_SRV;
```

Members

`MostDetailedMip`

Type: **UINT**

Index of the most detailed mipmap level to use; this number is between 0 and **MipLevels** -1.

`MipLevels`

Type: **UINT**

Number of mipmap levels to use.

Remarks

This structure is one member of a shader-resource-view description (see D3D10_SHADER_RESOURCE_VIEW_DESC).

Requirements

Header	d3d10.h

See also

Resource Structures

D3D10_TEXCUBE_ARRAY_SRV1 structure (d3d10_1.h)

Article • 04/02/20212 minutes to read

Specifies the subresource(s) from an array of cube textures to use in a shader-resource view.

Syntax

```C++
typedef struct D3D10_TEXCUBE_ARRAY_SRV1 {
  UINT MostDetailedMip;
  UINT MipLevels;
  UINT First2DArrayFace;
  UINT NumCubes;
} D3D10_TEXCUBE_ARRAY_SRV1;
```

Members

`MostDetailedMip`

Type: **UINT**

Index of the most detailed mipmap level to use; this number is between 0 and **MipLevels**.

`MipLevels`

Type: **UINT**

Number of mipmap levels to use.

`First2DArrayFace`

Type: **UINT**

Index of the first 2D texture to use.

`NumCubes`

Type: **UINT**

Number of cube textures in the array.

Remarks

This structure is one member of a shader-resource-view description (see D3D10_SHADER_RESOURCE_VIEW_DESC1).

This structure requires Windows Vista Service Pack 1.

Requirements

Header	d3d10_1.h (include D3D10_1Shader.h)

See also

Resource Structures

D3D10_TEXCUBE_SRV structure (d3d10.h)

Article07/27/2022

Specifies the subresource from a cube texture to use in a shader-resource view.

Syntax

C++

```
typedef struct D3D10_TEXCUBE_SRV {
  UINT MostDetailedMip;
  UINT MipLevels;
} D3D10_TEXCUBE_SRV;
```

Members

`MostDetailedMip`

Type: **UINT**

Index of the most detailed mipmap level to use; this number is between 0 and **MipLevels** -1.

`MipLevels`

Type: **UINT**

Number of mipmap levels to use.

Remarks

This structure is one member of a shader-resource-view description (see D3D10_SHADER_RESOURCE_VIEW_DESC).

Requirements

Header	d3d10.h

See also

Resource Structures

CD3D10_TEXTURE1D_DESC structure (d3d10.h)

Article • 02/16/20232 minutes to read

Describes a 1D texture.

Syntax

C++

```
struct CD3D10_TEXTURE1D_DESC : D3D10_TEXTURE1D_DESC {
  void CD3D10_TEXTURE1D_DESC();
  void CD3D10_TEXTURE1D_DESC(
    const D3D10_TEXTURE1D_DESC & o
  );
  void CD3D10_TEXTURE1D_DESC(
    DXGI_FORMAT format,
    UINT        width,
    UINT        arraySize,
    UINT        mipLevels,
    UINT        bindFlags,
    D3D10_USAGE usage,
    UINT        cpuaccessFlags,
    UINT        miscFlags
  );
  void ~CD3D10_TEXTURE1D_DESC();
};
```

Inheritance

The **CD3D10_TEXTURE1D_DESC** structure implements D3D10_TEXTURE1D_DESC.

Members

`void CD3D10_TEXTURE1D_DESC()`

`void CD3D10_TEXTURE1D_DESC( const D3D10_TEXTURE1D_DESC & o)`

`void CD3D10_TEXTURE1D_DESC( DXGI_FORMAT format, UINT width, UINT arraySize, UINT mipLevels, UINT bindFlags, D3D10_USAGE usage, UINT cpuaccessFlags, UINT miscFlags)`

`void ~CD3D10_TEXTURE1D_DESC()`

Remarks

`format` Type: **DXGI_FORMAT** Texture format (see DXGI_FORMAT).

`width` Type: **UINT** Texture width (in texels). The range is from 1 to D3D10_REQ_TEXTURE1D_U_DIMENSION (8192).

`arraySize` Type: **UINT** Number of textures in the array. The range is from 1 to D3D10_REQ_TEXTURE1D_ARRAY_AXIS_DIMENSION (512).

`mipLevels` Type: **UINT** Number of subtextures (also called mipmap levels). Use 1 for a multisampled texture; or 0 to generate a full set of subtextures.

`bindFlags` Type: **UINT** Flags (see D3D10_BIND_FLAG) for binding to pipeline stages. The flags can be combined by a logical OR.

`usage` Type: **D3D10_USAGE** Value that identifies how the texture is to be read from and written to. The most common value is D3D10_USAGE-DEFAULT; see D3D10_USAGE for all possible values.

`cpuAccessFlags` Type: **UINT** Flags (see D3D10_CPU_ACCESS_FLAG) to specify the types of CPU access allowed. Use 0 if CPU access is not required. These flags can be combined with a logical OR.

`miscFlags` Type: **UINT** Flags (see D3D10_RESOURCE_MISC_FLAG) that identify other, less common resource options. Use 0 if none of these flags apply. These flags can be combined with a logical OR.

This structure is used in a call to ID3D10Device::CreateTexture1D. A helpful derived structure CD3D10_TEXTURE1D_DESC is declared in D3D10.h, to help create a texture description.

Requirements

Header	d3d10.h

See also

Resource Structures

CD3D10_TEXTURE2D_DESC structure (d3d10.h)

Article • 02/16/20232 minutes to read

Describes a 2D texture.

Syntax

C++

```
struct CD3D10_TEXTURE2D_DESC : D3D10_TEXTURE2D_DESC {
  void CD3D10_TEXTURE2D_DESC();
  void CD3D10_TEXTURE2D_DESC(
    const D3D10_TEXTURE2D_DESC & o
  );
  void CD3D10_TEXTURE2D_DESC(
    DXGI_FORMAT format,
    UINT        width,
    UINT        height,
    UINT        arraySize,
    UINT        mipLevels,
    UINT        bindFlags,
    D3D10_USAGE usage,
    UINT        cpuaccessFlags,
    UINT        sampleCount,
    UINT        sampleQuality,
    UINT        miscFlags
  );
  void ~CD3D10_TEXTURE2D_DESC();
};
```

Inheritance

The **CD3D10_TEXTURE2D_DESC** structure implements D3D10_TEXTURE2D_DESC.

Members

`void CD3D10_TEXTURE2D_DESC()`

`void CD3D10_TEXTURE2D_DESC( const D3D10_TEXTURE2D_DESC & o)`

`void CD3D10_TEXTURE2D_DESC( DXGI_FORMAT format, UINT width, UINT height, UINT arraySize, UINT mipLevels, UINT bindFlags, D3D10_USAGE usage, UINT cpuaccessFlags,`

`UINT sampleCount, UINT sampleQuality, UINT miscFlags)`

`void ~CD3D10_TEXTURE2D_DESC()`

Remarks

`format` Type: **DXGI_FORMAT** Texture format (see DXGI_FORMAT).

`width` Type: **UINT** Texture width (in texels). The range is from 1 to D3D10_REQ_TEXTURE2D_U_OR_V_DIMENSION (8192). For a texture cube-map, the range is from 1 to D3D10_REQ_TEXTURECUBE_DIMENSION (8192). For more information about restrictions, see Remarks.

`height` Type: **UINT** Texture height (in texels). The range is from 1 to D3D10_REQ_TEXTURE2D_U_OR_V_DIMENSION (8192). For a texture cube-map, the range is from 1 to D3D10_REQ_TEXTURECUBE_DIMENSION (8192). For more information about restrictions, see Remarks.

`arraySize` Type: **UINT** Number of textures in the texture array. The range is from 1 to D3D10_REQ_TEXTURE2D_ARRAY_AXIS_DIMENSION (512). For a texture cube-map, this value is a multiple of 6 (that is, 6 * the value in the **NumCubes** member of D3D10_TEXCUBE_ARRAY_SRV1), and the range is from 6 to D3D10_REQ_TEXTURECUBE_DIMENSION.

`mipLevels` Type: **UINT** Number of subtextures (also called mipmap levels). Use 1 for a multisampled texture; or 0 to generate a full set of subtextures.

`bindFlags` Type: **UINT** Flags (see D3D10_BIND_FLAG) for binding to pipeline stages. The flags can be combined by a logical OR.

`usage` Type: **D3D10_USAGE** Value that identifies how the texture is to be read from and written to. The most common value is D3D10_USAGE-DEFAULT; see D3D10_USAGE for all possible values.

`cpuAccessFlags` Type: **UINT** Flags (see D3D10_CPU_ACCESS_FLAG) to specify the types of CPU access allowed. Use 0 if CPU access is not required. These flags can be combined with a logical OR.

`sampleCount` Type: **UINT**

`sampleQuality` Type: **UINT**

`miscFlags` Type: **UINT** Flags (see D3D10_RESOURCE_MISC_FLAG) that identify other, less common resource options. Use 0 if none of these flags apply. These flags can be combined with a logical OR. For a texture cube-map, set the D3D10_RESOURCE_MISC_TEXTURECUBE flag. Cube-map arrays (that is, **ArraySize** > 6) require feature level D3D_FEATURE_LEVEL_10_1.

This structure is used in a call to ID3D10Device::CreateTexture2D. A helpful derived structure CD3D10_TEXTURE2D_DESC is declared in D3D10.h, to help create a texture description.

The device places some size restrictions (must be multiples of a minimum size) for a subsampled, block compressed, or bit-format resource.

Requirements

Header	d3d10.h

See also

Resource Structures

CD3D10_TEXTURE3D_DESC structure (d3d10.h)

Article • 02/16/20232 minutes to read

Describes a 3D texture.

Syntax

C++

```
struct CD3D10_TEXTURE3D_DESC : D3D10_TEXTURE3D_DESC {
  void CD3D10_TEXTURE3D_DESC();
  void CD3D10_TEXTURE3D_DESC(
    const D3D10_TEXTURE3D_DESC & o
  );
  void CD3D10_TEXTURE3D_DESC(
    DXGI_FORMAT format,
    UINT        width,
    UINT        height,
    UINT        depth,
    UINT        mipLevels,
    UINT        bindFlags,
    D3D10_USAGE usage,
    UINT        cpuaccessFlags,
    UINT        miscFlags
  );
  void ~CD3D10_TEXTURE3D_DESC();
};
```

Inheritance

The **CD3D10_TEXTURE3D_DESC** structure implements D3D10_TEXTURE3D_DESC.

Members

`void CD3D10_TEXTURE3D_DESC()`

`void CD3D10_TEXTURE3D_DESC( const D3D10_TEXTURE3D_DESC & o)`

`void CD3D10_TEXTURE3D_DESC( DXGI_FORMAT format, UINT width, UINT height, UINT depth, UINT mipLevels, UINT bindFlags, D3D10_USAGE usage, UINT cpuaccessFlags, UINT miscFlags)`

`void ~CD3D10_TEXTURE3D_DESC()`

Remarks

`format` Type: **DXGI_FORMAT** Texture format (see DXGI_FORMAT).

`width` Type: **UINT** Texture width (in texels). The range is from 1 to D3D10_REQ_TEXTURE3D_U_V_OR_W_DIMENSION (2048). For more information about restrictions, see Remarks.

`height` Type: **UINT** Texture height (in texels). The range is from 1 to D3D10_REQ_TEXTURE3D_U_V_OR_W_DIMENSION (2048). For more information about restrictions, see Remarks.

`depth` Type: **UINT** Texture depth (in texels). The range is from 1 to D3D10_REQ_TEXTURE3D_U_V_OR_W_DIMENSION (2048).

`mipLevels` Type: **UINT** Number of subtextures (also called mipmap levels). Use 1 for a multisampled texture; or 0 to generate a full set of subtextures.

`bindFlags` Type: **UINT** Flags (see D3D10_BIND_FLAG) for binding to pipeline stages. The flags can be combined by a logical OR.

`usage` Type: **D3D10_USAGE** Value that identifies how the texture is to be read from and written to. The most common value is D3D10_USAGE-DEFAULT; see D3D10_USAGE for all possible values.

`cpuAccessFlags` Type: **UINT** Flags (see D3D10_CPU_ACCESS_FLAG) to specify the types of CPU access allowed. Use 0 if CPU access is not required. These flags can be combined with a logical OR.

`miscFlags` Type: **UINT** Flags (see D3D10_RESOURCE_MISC_FLAG) that identify other, less common resource options. Use 0 if none of these flags apply. These flags can be combined with a logical OR.

This structure is used in a call to ID3D10Device::CreateTexture3D. A helpful derived structure CD3D10_TEXTURE3D_DESC is declared in D3D10.h, to help create a texture description.

The device restricts the size of subsampled, block compressed (see Block Compression (Direct3D 10)), and bit format resources to be multiples of sizes specific to each format.

Requirements

Header	d3d10.h

See also

Resource structures

Resource Enumerations (Direct3D 10 Graphics)

Article • 01/06/2021 • 2 minutes to read

The following enumerations are used to specify information about how resources are created and accessed during rendering.

Enumeration	Description
D3D10_BIND_FLAG	Identifies how a resource will be used by the pipeline and determines which pipeline stage(s) a resource can be bound to.
D3D10_CPU_ACCESS_FLAG	Specifies the types of CPU access allowed for a resource.
D3D10_DSV_DIMENSION	Specifies how to access a resource that is used in a depth-stencil view.
D3D10_MAP	Identifies a resource to be mapped for reading and writing by the CPU.
D3D10_MAP_FLAG	Specifies how the CPU should respond to any Map methods when the GPU is not yet finished with the resource.
D3D10_RESOURCE_DIMENSION	Identifies the type of resource that is in use.
D3D10_RESOURCE_MISC_FLAG	Identifies less common resource creation options.
D3D10_RTV_DIMENSION	Specifies how to access a resource used in a render target view.
D3D10_USAGE	Identifies how a resource is expected to be read and written by the CPU and/or the GPU.

Related topics

Resource Reference

D3D10_BIND_FLAG enumeration (d3d10.h)

Article • 07/27/20222 minutes to read

Identifies how to bind a resource to the pipeline.

Syntax

C++

```
typedef enum D3D10_BIND_FLAG {
  D3D10_BIND_VERTEX_BUFFER = 0x1L,
  D3D10_BIND_INDEX_BUFFER = 0x2L,
  D3D10_BIND_CONSTANT_BUFFER = 0x4L,
  D3D10_BIND_SHADER_RESOURCE = 0x8L,
  D3D10_BIND_STREAM_OUTPUT = 0x10L,
  D3D10_BIND_RENDER_TARGET = 0x20L,
  D3D10_BIND_DEPTH_STENCIL = 0x40L
} ;
```

Constants

`D3D10_BIND_VERTEX_BUFFER` Value: *0x1L* Bind a buffer as a vertex buffer to the input-assembler stage.
`D3D10_BIND_INDEX_BUFFER` Value: *0x2L* Bind a buffer as an index buffer to the input-assembler stage.
`D3D10_BIND_CONSTANT_BUFFER` Value: *0x4L* Bind a buffer as a constant buffer to a shader stage; this flag may NOT be combined with any other bind flag.
`D3D10_BIND_SHADER_RESOURCE` Value: *0x8L* Bind a buffer or texture to a shader stage; this flag cannot be used with the D3D10_MAP_WRITE_NO_OVERWRITE flag.

`D3D10_BIND_STREAM_OUTPUT` Value: *0x10L* Bind an output buffer for the stream-output stage.	
`D3D10_BIND_RENDER_TARGET` Value: *0x20L* Bind a texture as a render target for the output-merger stage.	
`D3D10_BIND_DEPTH_STENCIL` Value: *0x40L* Bind a texture as a depth-stencil target for the output-merger stage.	

Remarks

In general, binding flags can be combined using a logical OR (except the constant-buffer flag); however, you should use a single flag to allow the device to optimize the resource usage.

This enumeration is used by a:

- Buffer description when creating a buffer.
- Texture description when creating a texture (see D3D10_TEXTURE1D_DESC or D3D10_TEXTURE2D_DESC or D3D10_TEXTURE3D_DESC).

A shader-resource buffer is NOT a constant buffer; rather, it is a texture or buffer resource that is bound to a shader, that contains texture or buffer data (it is not limited to a single element type in the buffer). A shader-resource buffer is created with the D3D10_BIND_SHADER_RESOURCE flag and is bound to the pipeline using one of these APIs: ID3D10Device::GSSetShaderResources, ID3D10Device::PSSetShaderResources, or ID3D10Device::VSSetShaderResources. Furthermore, a shader-resource buffer cannot use the D3D10_MAP_WRITE_NO_OVERWRITE flag.

Requirements

Header	d3d10.h

See also

Resource Enumerations

D3D10_CPU_ACCESS_FLAG enumeration (d3d10.h)

Article • 07/27/20222 minutes to read

Specifies the types of CPU access allowed for a resource.

Syntax

C++

```
typedef enum D3D10_CPU_ACCESS_FLAG {
  D3D10_CPU_ACCESS_WRITE = 0x10000L,
  D3D10_CPU_ACCESS_READ = 0x20000L
} ;
```

Constants

`D3D10_CPU_ACCESS_WRITE` Value: *0x10000L* The resource is to be mappable so that the CPU can change its contents. Resources created with this flag cannot be set as outputs of the pipeline and must be created with either dynamic or staging usage (see D3D10_USAGE).
`D3D10_CPU_ACCESS_READ` Value: *0x20000L* The resource is to be mappable so that the CPU can read its contents. Resources created with this flag cannot be set as either inputs or outputs to the pipeline and must be created with staging usage (see D3D10_USAGE).

Remarks

This enumeration is used in D3D10_BUFFER_DESC, D3D10_TEXTURE1D_DESC, D3D10_TEXTURE2D_DESC, D3D10_TEXTURE3D_DESC, and D3DX10_IMAGE_LOAD_INFO. See Creating Buffer Resources (Direct3D 10) for more details.

Applications can combine one or more of these flags with a bitwise OR. When possible, create resources with no CPU access flags, as this enables better resource optimization.

Requirements

Header	d3d10.h

See also

Resource Enumerations

D3D10_DSV_DIMENSION enumeration (d3d10.h)

Article • 07/27/20222 minutes to read

Specifies how to access a resource used in a depth-stencil view.

Syntax

C++

```
typedef enum D3D10_DSV_DIMENSION {
  D3D10_DSV_DIMENSION_UNKNOWN = 0,
  D3D10_DSV_DIMENSION_TEXTURE1D = 1,
  D3D10_DSV_DIMENSION_TEXTURE1DARRAY = 2,
  D3D10_DSV_DIMENSION_TEXTURE2D = 3,
  D3D10_DSV_DIMENSION_TEXTURE2DARRAY = 4,
  D3D10_DSV_DIMENSION_TEXTURE2DMS = 5,
  D3D10_DSV_DIMENSION_TEXTURE2DMSARRAY = 6
} ;
```

Constants

`D3D10_DSV_DIMENSION_UNKNOWN` Value: *0* The resource will be accessed according to its type as determined from the actual instance this enumeration is paired with when the depth-stencil view is created.
`D3D10_DSV_DIMENSION_TEXTURE1D` Value: *1* The resource will be accessed as a 1D texture.
`D3D10_DSV_DIMENSION_TEXTURE1DARRAY` Value: *2* The resource will be accessed as an array of 1D textures.
`D3D10_DSV_DIMENSION_TEXTURE2D` Value: *3* The resource will be accessed as a 2D texture.
`D3D10_DSV_DIMENSION_TEXTURE2DARRAY` Value: *4* The resource will be accessed as an array of 2D texture.

`D3D10_DSV_DIMENSION_TEXTURE2DMS` Value: *5* The resource will be accessed as a 2D texture with multisampling.	
`D3D10_DSV_DIMENSION_TEXTURE2DMSARRAY` Value: *6* The resource will be accessed as an array of 2D textures with multisampling.	

Remarks

This enumeration is used in D3D10_DEPTH_STENCIL_VIEW_DESC to create a depth-stencil view.

Requirements

Header	d3d10.h

See also

Resource Enumerations

D3D10_MAP enumeration (d3d10.h)

Article07/27/2022

Identifies a resource to be accessed for reading and writing by the CPU. Applications may combine one or more of these flags.

Syntax

C++

```
typedef enum D3D10_MAP {
  D3D10_MAP_READ = 1,
  D3D10_MAP_WRITE = 2,
  D3D10_MAP_READ_WRITE = 3,
  D3D10_MAP_WRITE_DISCARD = 4,
  D3D10_MAP_WRITE_NO_OVERWRITE = 5
} ;
```

Constants

`D3D10_MAP_READ` Value: *1* Resource is mapped for reading. The resource must have been created with read access (see D3D10_CPU_ACCESS_READ).
`D3D10_MAP_WRITE` Value: *2* Resource is mapped for writing. The resource must have been created with write access (see D3D10_CPU_ACCESS_WRITE).
`D3D10_MAP_READ_WRITE` Value: *3* Resource is mapped for reading and writing. The resource must have been created with read and write access (see D3D10_CPU_ACCESS_READ and D3D10_CPU_ACCESS_WRITE).
`D3D10_MAP_WRITE_DISCARD` Value: *4* Resource is mapped for writing; the previous contents of the resource will be undefined. The resource must have been created with write access (see D3D10_CPU_ACCESS_WRITE).

`D3D10_MAP_WRITE_NO_OVERWRITE` Value: *5* Resource is mapped for writing; the existing contents of the resource cannot be overwritten (see Remarks). This flag is only valid on vertex and index buffers. The resource must have been created with write access (see D3D10_CPU_ACCESS_WRITE). Cannot be used on a resource created with the D3D10_BIND_CONSTANT_BUFFER flag.	

Remarks

This enumeration is used in ID3D10Buffer::Map, ID3D10Texture1D::Map, ID3D10Texture2D::Map, and ID3D10Texture3D::Map.

Meaning of D3D10_MAP_WRITE_NO_OVERWRITE

D3D10_MAP_WRITE_NO_OVERWRITE signifies that the application promises not to write to data that the Input Assembler (IA) stage is using. In exchange, the GPU allows the application to write to other parts of the same buffer. The application must ensure that it does not write over any data in use by the IA stage.

For example, consider the buffer illustrated in the following diagram. If a Draw call has been issued that uses vertices 4-6, an application that calls Map on this buffer must ensure that it does not write to the vertices that the Draw call will access during rendering.

However, ensuring this can be difficult, because the GPU is often many frames behind the CPU in terms of which frame it is currently processing. Keeping track of which sections of a resource are being used because of calls made 2 to 5 frames ago is difficult and error-prone. Because of this, it is recommended that applications only write to the uninitialized portions of a resource when using D3D10_MAP_WRITE_NO_OVERWRITE.

Common Usage of D3D10_MAP_WRITE_DISCARD with D3D10_MAP_WRITE_NO_OVERWRITE

D3D10_MAP_WRITE_DISCARD and D3D10_MAP_WRITE_NO_OVERWRITE are normally used in conjunction with dynamic index/vertex buffers, although they can also be used with dynamic textures.

A common use of these two flags involves filling dynamic index/vertex buffers with geometry that can be seen from the camera's current position. The first time that data is entered into the buffer on a given frame, Map is called with D3D10_MAP_WRITE_DISCARD; doing so invalidates the previous contents of the buffer. The buffer is then filled with all available data.

Subsequent writes to the buffer within the same frame should use D3D10_MAP_WRITE_NO_OVERWRITE. This will enable the CPU to access a resource that is potentially being used by the GPU as long as the restrictions described previously are respected.

Requirements

Header	d3d10.h

See also

Resource Enumerations

D3D10_MAP_FLAG enumeration (d3d10.h)

Article01/31/2022

Specifies how the CPU should respond when Map is called on a resource being used by the GPU.

Syntax

C++

```
typedef enum D3D10_MAP_FLAG {
  D3D10_MAP_FLAG_DO_NOT_WAIT = 0x100000L
} ;
```

Constants

`D3D10_MAP_FLAG_DO_NOT_WAIT` Value: *0x100000L* Specifies that Map should return **DXGI_ERROR_WAS_STILL_DRAWING** when the GPU blocks the CPU from accessing a resource.

Remarks

This enumeration is used by ID3D10Buffer::Map, ID3D10Texture1D::Map, ID3D10Texture2D::Map, and ID3D10Texture3D::Map.

D3D10_MAP_FLAG_DO_NOT_WAIT cannot be used with D3D10_MAP_WRITE_DISCARD or D3D10_MAP_WRITE_NOOVERWRITE.

For more information about potential conflicts between the GPU and CPU during resource mapping, see Copying and Accessing Resource Data (Direct3D 10).

Requirements

Header	d3d10.h

See also

Resource Enumerations

D3D10_RESOURCE_DIMENSION enumeration (d3d10.h)

Article07/27/2022

Identifies the type of resource being used.

Syntax

C++

```
typedef enum D3D10_RESOURCE_DIMENSION {
  D3D10_RESOURCE_DIMENSION_UNKNOWN = 0,
  D3D10_RESOURCE_DIMENSION_BUFFER = 1,
  D3D10_RESOURCE_DIMENSION_TEXTURE1D = 2,
  D3D10_RESOURCE_DIMENSION_TEXTURE2D = 3,
  D3D10_RESOURCE_DIMENSION_TEXTURE3D = 4
} ;
```

Constants

`D3D10_RESOURCE_DIMENSION_UNKNOWN` Value: *0* Resource is of unknown type.
`D3D10_RESOURCE_DIMENSION_BUFFER` Value: *1* Resource is a buffer.
`D3D10_RESOURCE_DIMENSION_TEXTURE1D` Value: *2* Resource is a 1D texture.
`D3D10_RESOURCE_DIMENSION_TEXTURE2D` Value: *3* Resource is a 2D texture.
`D3D10_RESOURCE_DIMENSION_TEXTURE3D` Value: *4* Resource is a 3D texture.

Remarks

This enumeration is used in ID3D10Resource::GetType, and D3DX10_IMAGE_INFO.

Requirements

Header	d3d10.h

See also

Resource Enumerations

D3D10_RESOURCE_MISC_FLAG enumeration (d3d10.h)

Article01/31/2022

Identifies other, less common options for resources.

Syntax

C++

```
typedef enum D3D10_RESOURCE_MISC_FLAG {
  D3D10_RESOURCE_MISC_GENERATE_MIPS = 0x1L,
  D3D10_RESOURCE_MISC_SHARED = 0x2L,
  D3D10_RESOURCE_MISC_TEXTURECUBE = 0x4L,
  D3D10_RESOURCE_MISC_SHARED_KEYEDMUTEX = 0x10L,
  D3D10_RESOURCE_MISC_GDI_COMPATIBLE = 0x20L
} ;
```

Constants

`D3D10_RESOURCE_MISC_GENERATE_MIPS` Value: *0x1L* Enables an application to call ID3D10Device::GenerateMips on a texture resource. The resource must be created with the bind flags that specify that the resource is a render target and a shader resource.
`D3D10_RESOURCE_MISC_SHARED` Value: *0x2L* Enables the sharing of resource data between two or more Direct3D devices. The only resources that can be shared are 2D non-mipmapped textures. WARP and REF devices do not support shared resources. Attempting to create a resource with this flag on either a WARP or REF device will cause the create method to return an E_OUTOFMEMORY error code.
`D3D10_RESOURCE_MISC_TEXTURECUBE` Value: *0x4L* Enables an application to create a cube texture from a Texture2DArray that contains 6 textures.

`D3D10_RESOURCE_MISC_SHARED_KEYEDMUTEX` Value: *0x10L* Enables the resource created to be synchronized using the IDXGIKeyedMutex::AcquireSync and ReleaseSync APIs. The following resource creation D3D10 APIs, that all take a D3D10_RESOURCE_MISC_FLAG parameter, have been extended to support the new flag. • ID3D10Device1::CreateTexture1D • ID3D10Device1::CreateTexture2D • ID3D10Device1::CreateTexture3D • ID3D10Device1::CreateBuffer If any of the listed functions are called with the D3D10_RESOURCE_MISC_SHARED_KEYEDMUTEX flag set, the interface returned can be queried for an IDXGIKeyedMutex interface, which implements AcquireSync and ReleaseSync APIs to synchronize access to the surface. The device creating the surface, and any other device opening the surface (using OpenSharedResource) is required to call IDXGIKeyedMutex::AcquireSync before any rendering commands to the surface, and IDXGIKeyedMutex::ReleaseSync when it is done rendering. WARP and REF devices do not support shared resources. Attempting to create a resource with this flag on either a WARP or REF device will cause the create method to return an E_OUTOFMEMORY error code.	
`D3D10_RESOURCE_MISC_GDI_COMPATIBLE` Value: *0x20L* Enables a surface to be used for GDI interoperability. Setting this flag enables rendering on the surface via IDXGISurface1::GetDC.	

Remarks

This enumeration is used in D3D10_BUFFER_DESC, D3D10_TEXTURE1D_DESC, D3D10_TEXTURE2D_DESC, D3D10_TEXTURE3D_DESC, D3DX10_IMAGE_INFO, and D3DX10_IMAGE_LOAD_INFO.

These flags can be combined by bitwise OR.

D3D10_RESOURCE_MISC_SHARED and D3D10_RESOURCE_MISC_SHARED_KEYEDMUTEX are mutually exclusive flags: either one may be set in the resource creation calls but not both simultaneously.

Requirements

Header	d3d10.h

See also

Resource Enumerations

D3D10_RTV_DIMENSION enumeration (d3d10.h)

Article • 01/31/20222 minutes to read

Specifies how to access a resource used in a render-target view.

Syntax

C++

```
typedef enum D3D10_RTV_DIMENSION {
  D3D10_RTV_DIMENSION_UNKNOWN = 0,
  D3D10_RTV_DIMENSION_BUFFER = 1,
  D3D10_RTV_DIMENSION_TEXTURE1D = 2,
  D3D10_RTV_DIMENSION_TEXTURE1DARRAY = 3,
  D3D10_RTV_DIMENSION_TEXTURE2D = 4,
  D3D10_RTV_DIMENSION_TEXTURE2DARRAY = 5,
  D3D10_RTV_DIMENSION_TEXTURE2DMS = 6,
  D3D10_RTV_DIMENSION_TEXTURE2DMSARRAY = 7,
  D3D10_RTV_DIMENSION_TEXTURE3D = 8
} ;
```

Constants

`D3D10_RTV_DIMENSION_UNKNOWN` Value: *0* The resource will be accessed according to its type as determined from the actual instance this enumeration is paired with when the render-target view is created.
`D3D10_RTV_DIMENSION_BUFFER` Value: *1* The resource will be accessed as a buffer.
`D3D10_RTV_DIMENSION_TEXTURE1D` Value: *2* The resource will be accessed as a 1D texture.
`D3D10_RTV_DIMENSION_TEXTURE1DARRAY` Value: *3* The resource will be accessed as an array of 1D textures.

`D3D10_RTV_DIMENSION_TEXTURE2D` Value: *4* The resource will be accessed as a 2D texture.	
`D3D10_RTV_DIMENSION_TEXTURE2DARRAY` Value: *5* The resource will be accessed as an array of 2D textures.	
`D3D10_RTV_DIMENSION_TEXTURE2DMS` Value: *6* The resource will be accessed as a 2D texture with multisampling.	
`D3D10_RTV_DIMENSION_TEXTURE2DMSARRAY` Value: *7* The resource will be accessed as an array of 2D textures with multisampling.	
`D3D10_RTV_DIMENSION_TEXTURE3D` Value: *8* The resource will be accessed as a 3D texture.	

Remarks

This enumeration is used in D3D10_RENDER_TARGET_VIEW_DESC to create a render-target view.

Requirements

Header	d3d10.h

See also

Resource Enumerations

D3D10_USAGE enumeration (d3d10.h)

Article01/31/2022

Identifies expected resource use during rendering. The usage directly reflects whether a resource is accessible by the CPU and/or the GPU.

Syntax

C++

```
typedef enum D3D10_USAGE {
  D3D10_USAGE_DEFAULT = 0,
  D3D10_USAGE_IMMUTABLE = 1,
  D3D10_USAGE_DYNAMIC = 2,
  D3D10_USAGE_STAGING = 3
} ;
```

Constants

`D3D10_USAGE_DEFAULT` Value: *0* A resource that requires read and write access by the GPU. This is likely to be the most common usage choice.
`D3D10_USAGE_IMMUTABLE` Value: *1* A resource that can only be read by the GPU. It cannot be written by the GPU, and cannot be accessed at all by the CPU. This type of resource must be initialized when it is created, since it cannot be changed after creation.
`D3D10_USAGE_DYNAMIC` Value: *2* A resource that is accessible by both the GPU and the CPU (write only). A dynamic resource is a good choice for a resource that will be updated by the CPU at least once per frame. To write to a dynamic resource on the CPU, use a **Map** method. You can write to a dynamic resource on the GPU using CopyResource or CopySubresourceRegion.
`D3D10_USAGE_STAGING` Value: *3* A resource that supports data transfer (copy) from the GPU to the CPU.

Remarks

An application identifies the way a resource is intended to be used (its usage) in a resource description. There are several structures for creating resources including: D3D10_TEXTURE1D_DESC, D3D10_TEXTURE2D_DESC, D3D10_TEXTURE3D_DESC, D3D10_BUFFER_DESC, and D3DX10_IMAGE_LOAD_INFO.

> Differences between Direct3D 9 and Direct3D 10:
> In Direct3D 9, you specify the type of memory a resource should be created in at resource creation time (using D3DPOOL). It was an application's job to decide what memory pool would provide the best combination of functionality and performance.
>
> In Direct3D 10, an application no longer specifies what type of memory (the pool) to create a resource in. Instead, you specify the intended usage of the resource, and let the runtime (in concert with the driver and a memory manager) choose the type of memory that will achieve the best performance.

Resource Usage Restrictions

Each usage dictates a tradeoff between functionality and performance. In general, resource accessing is accomplished with the following APIs.

- CPU access is done with ID3D10Buffer::Map, ID3D10Texture1D::Map, ID3D10Texture2D::Map, or ID3D10Texture3D::Map
- GPU access is done with CopySubresourceRegion, CopyResource, or UpdateSubresource.

Use the following table to choose the usage that best describes how the resource will need to be accessed by the CPU and/or the GPU. Of course, there will be performance tradeoffs.

Resource Usage	Default	Dynamic	Immutable	Staging
GPU-Read	yes	yes[1]	yes	yes[1, 2]
GPU-Write	yes[1]			yes[1, 2]
CPU-Read				yes[1, 2]
CPU-Write		yes		yes[1, 2]

- 1 - This is restricted to CopySubresourceRegion, CopyResource, and UpdateSubresource.

- 2 - Cannot be a depth-stencil buffer or a multisampled render target.

Resource Bind Options

To maximize performance, not all resource usage options can be used as input or output resources to the pipeline. This table identifies these limitations.

Resource Can Be Bound As	Default	Dynamic	Immutable	Staging
Input to a Stage	yes[3]	yes[4]	yes	
Output from a Stage	yes[3]			

- 3 - If bound as an input and an output using different views, each view must use different subresources.
- 4 - The resource can only be created with a single subresource. The resource cannot be a texture array. The resource cannot be a mipmap chain.

Requirements

Header	d3d10.h

See also

Resource Enumerations

Resource Constants

Article • 01/06/2021 • 2 minutes to read

The following constants are used to specify information about how resources are created and accessed during rendering.

#define	Description
D3D10_REQ	Resource limitation constants.

Related topics

Resource Reference

D3D10_REQ

Article • 01/06/2021 • 2 minutes to read

Resource limitation constants.

#define	Value	Description
D3D10_REQ_MIP_LEVELS	14	Maximum number of mipmap levels a texture resource may have.
D3D10_REQ_RESOURCE_SIZE_IN_MEGABYTES	128	Maximum possible size of a resource in megabytes. Apps can create resources bigger than the maximum resource size on some graphics hardware. However, we recommend that apps keep resources smaller than the maximum resource size to get the maximum amount of compatibility across graphics vendors. The runtime only guarantees that allocations within the maximum resource size are supported by all Direct3D 10 hardware.
D3D10_REQ_TEXTURE1D_ARRAY_AXIS_DIMENSION	512	Maximum array size of a 1D texture array.
D3D10_REQ_TEXTURE2D_ARRAY_AXIS_DIMENSION	512	Maximum array size of a 2D texture array.
D3D10_REQ_TEXTURE1D_U_DIMENSION	8192	Maximum width of a 1D texture.
D3D10_REQ_TEXTURE2D_U_OR_V_DIMENSION	8192	Maximum width and height of a 2D texture.
D3D10_REQ_TEXTURE3D_U_V_OR_W_DIMENSION	2048	Maximum width, height, and depth of a 3D texture.
D3D10_REQ_TEXTURECUBE_DIMENSION	8192	Maximum size of a cube texture.

The constants are defined in D3D10.h.

Related topics

Resource Constants

Direct3D Reference

Shader Reference (Direct3D 10 Graphics)

Article • 01/06/2021 • 2 minutes to read

The Direct3D API defines several API elements to help you create and manage programmable shaders.

- Shader Interfaces
- Shader Functions
- Shader Structures
- Shader Enumerations
- Shader Constants

Shaders are executable programs that control the vertex shader, geometry shader and pixel shader pipeline stages. Shaders are programmed exclusively using HLSL.

Related topics

Direct3D Reference

Shader Interfaces (Direct3D 10 Graphics)

Article • 01/06/2021 • 2 minutes to read

This section contains information about the following 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.

Pipeline-Stage Interfaces	Description
ID3D10GeometryShader Interface	A geometry-shader implements per-primitive processing in the geometry-shader stage.
ID3D10PixelShader Interface	A pixel-shader implements per-pixel processing in the pixel-shader stage.
ID3D10VertexShader Interface	A vertex-shader implements per-vertex processing in the vertex-shader stage.

Shader-reflection interfaces allow an application to inspect the contents of a shader at design/author time. Shader reflection is not useful for setting variables at runtime as it is a mirror of the shader data, and therefore supports no methods for setting data.

Shader-Reflection Interfaces	Description
ID3D10ShaderReflection Interface	A COM interface for reading information from a compiled shader at author time.
ID3D10ShaderReflectionConstantBuffer Interface	A helper interface for getting a shader-reflection constant-buffer interface.
ID3D10ShaderReflectionType Interface	A helper interface for getting a shader-reflection-type interface.
ID3D10ShaderReflectionVariable Interface	A helper interface for getting a shader-reflection-variable interface.
ID3D10ShaderResourceView Interface	A shader-reflection interface for reading information from a shader-resource view.

Shader reflection APIs implement one COM shader reflection interface (**ID3D10ShaderReflection Interface**) and several non-COM helper interfaces (the rest of the interfaces). **ID3D10ShaderReflection Interface** is created when a shader reflection object is created. It follows standard COM rules; creating the interface increases a reference count and the interface must be released when it is no longer needed. The remaining shader-reflection interfaces are helper interfaces that do not inherit from IUnknown. This means that they do not change any reference count when they are created, and they do not need to be destroyed when you are finished with them.

Related topics

Shader Reference

ID3D10GeometryShader interface (d3d10.h)

Article • 02/16/20232 minutes to read

A geometry-shader interface manages an executable program (a geometry shader) that controls the geometry-shader stage.

Inheritance

The **ID3D10GeometryShader** interface inherits from the ID3D10DeviceChild interface.

Remarks

The geometry-shader interface has no methods; use HLSL to implement your shader functionality. All shaders in Direct3D 10 are implemented from a common set of features referred to as the common shader core.

To create a geometry shader interface, call either ID3D10Device::CreateGeometryShader or ID3D10Device::CreateGeometryShaderWithStreamOutput. Before using a geometry shader you must bind it to the device by calling ID3D10Device::GSSetShader.

This interface is defined in D3D10.h.

Requirements

Target Platform	Windows
Header	d3d10.h

See also

ID3D10DeviceChild

Shader Interfaces

ID3D10PixelShader interface (d3d10.h)

Article • 02/16/20232 minutes to read

A pixel-shader interface manages an executable program (a pixel shader) that controls the pixel-shader stage.

Inheritance

The **ID3D10PixelShader** interface inherits from the ID3D10DeviceChild interface.

Remarks

The pixel-shader interface has no methods; use HLSL to implement your shader functionality. All shaders in Direct3D 10 are implemented from a common set of features referred to as the common shader core.

To create a pixel shader interface, call ID3D10Device::CreatePixelShader. Before using a pixel shader you must bind it to the device by calling ID3D10Device::PSSetShader.

This interface is defined in D3D10.h.

Requirements

Target Platform	Windows
Header	d3d10.h

See also

ID3D10DeviceChild

Shader Interfaces

ID3D10ShaderReflection interface (d3d10shader.h)

Article • 08/19/20222 minutes to read

A shader-reflection interface accesses shader information.

Inheritance

The **ID3D10ShaderReflection** interface inherits from the IUnknown interface. **ID3D10ShaderReflection** also has these types of members:

Methods

The **ID3D10ShaderReflection** interface has these methods.

ID3D10ShaderReflection::GetConstantBufferByIndex The ID3D10ShaderReflection::GetConstantBufferByIndex (d3d10shader.h) method gets a constant buffer by index.
ID3D10ShaderReflection::GetConstantBufferByName Get a constant buffer by name. (ID3D10ShaderReflection.GetConstantBufferByName)
ID3D10ShaderReflection::GetDesc Get a shader description. (ID3D10ShaderReflection.GetDesc)
ID3D10ShaderReflection::GetInputParameterDesc Get an input-parameter description for a shader. (ID3D10ShaderReflection.GetInputParameterDesc)
ID3D10ShaderReflection::GetOutputParameterDesc Get an output-parameter description for a shader. (ID3D10ShaderReflection.GetOutputParameterDesc)
ID3D10ShaderReflection::GetResourceBindingDesc Get a description of the resources bound to a shader.

Remarks

Create the interface by calling D3DX10ReflectShader. Since it is a COM interface, creating the interface increases a reference count and the interface must be released when it is no longer needed. The remaining shader-reflection interfaces are not COM interfaces.

Requirements

Target Platform	Windows
Header	d3d10shader.h

See also

Shader Interfaces

ID3D10ShaderReflection::GetConstantBufferByIndex method (d3d10shader.h)

Article • 08/19/20222 minutes to read

Get a constant buffer by index.

Syntax

C++

```
ID3D10ShaderReflectionConstantBuffer * GetConstantBufferByIndex(
  [in] UINT Index
);
```

Parameters

`[in] Index`

Type: **UINT**

Zero-based index.

Return value

Type: **ID3D10ShaderReflectionConstantBuffer***

A pointer to a constant buffer (see ID3D10ShaderReflectionConstantBuffer Interface).

Remarks

A constant buffer supplies either scalar constants or texture constants to a shader. A shader can use one or more constant buffers. For best performance, separate constants into buffers based on the frequency they are updated.

Requirements

Target Platform	Windows
Header	d3d10shader.h

See also

ID3D10ShaderReflection Interface

ID3D10ShaderReflection::GetConstantBufferByName method (d3d10shader.h)

Article07/27/2022

Get a constant buffer by name.

Syntax

```
C++

ID3D10ShaderReflectionConstantBuffer * GetConstantBufferByName(
  [in] LPCSTR Name
);
```

Parameters

`[in] Name`

Type: **LPCSTR**

The constant-buffer name.

Return value

Type: **ID3D10ShaderReflectionConstantBuffer***

A pointer to a constant buffer (see ID3D10ShaderReflectionConstantBuffer Interface).

Remarks

A constant buffer supplies either scalar constants or texture constants to a shader. A shader can use one or more constant buffers. For best performance, separate constants into buffers based on the frequency they are updated.

Requirements

Target Platform	Windows
Header	d3d10shader.h

See also

ID3D10ShaderReflection Interface

ID3D10ShaderReflection::GetDesc method (d3d10shader.h)

Article • 07/27/20222 minutes to read

Get a shader description.

Syntax

```
C++

HRESULT GetDesc(
  [in] D3D10_SHADER_DESC *pDesc
);
```

Parameters

`[in] pDesc`

Type: **D3D10_SHADER_DESC***

A pointer to a shader description. See D3D10_SHADER_DESC.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10shader.h

See also

ID3D10ShaderReflection Interface

ID3D10ShaderReflection::GetInputParameterDesc method (d3d10shader.h)

Article • 07/27/20222 minutes to read

Get an input-parameter description for a shader.

Syntax

C++

```
HRESULT GetInputParameterDesc(
  [in] UINT                           ParameterIndex,
  [in] D3D10_SIGNATURE_PARAMETER_DESC *pDesc
);
```

Parameters

`[in] ParameterIndex`

Type: **UINT**

A zero-based parameter index.

`[in] pDesc`

Type: **D3D10_SIGNATURE_PARAMETER_DESC***

A pointer to a shader-input-signature description. See D3D10_SIGNATURE_PARAMETER_DESC.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Remarks

An input-parameter description is also called a shader signature. The shader signature contains information about the input parameters such as the order or parameters, their

data type, and a parameter semantic.

Requirements

Target Platform	Windows
Header	d3d10shader.h

See also

ID3D10ShaderReflection Interface

ID3D10ShaderReflection::GetOutputParameterDesc method (d3d10shader.h)

Article • 07/27/20222 minutes to read

Get an output-parameter description for a shader.

Syntax

```
C++

HRESULT GetOutputParameterDesc(
  [in] UINT                          ParameterIndex,
  [in] D3D10_SIGNATURE_PARAMETER_DESC *pDesc
);
```

Parameters

`[in] ParameterIndex`

Type: **UINT**

A zero-based parameter index.

`[in] pDesc`

Type: **D3D10_SIGNATURE_PARAMETER_DESC***

A pointer to a shader-output-parameter description. See D3D10_SIGNATURE_PARAMETER_DESC.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Remarks

An output-parameter description is also called a shader signature. The shader signature contains information about the output parameters such as the order or parameters,

their data type, and a parameter semantic.

Requirements

Target Platform	Windows
Header	d3d10shader.h

See also

ID3D10ShaderReflection Interface

ID3D10ShaderReflection::GetResourceBindingDesc method (d3d10shader.h)

Article • 10/13/20212 minutes to read

Get a description of the resources bound to a shader.

Syntax

C++

```
HRESULT GetResourceBindingDesc(
  [in] UINT                          ResourceIndex,
  [in] D3D10_SHADER_INPUT_BIND_DESC *pDesc
);
```

Parameters

`[in] ResourceIndex`

Type: **UINT**

A zero-based resource index.

`[in] pDesc`

Type: **D3D10_SHADER_INPUT_BIND_DESC***

A pointer to an input-binding description. See D3D10_SHADER_INPUT_BIND_DESC.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Remarks

A shader consists of executable code (the compiled HLSL functions) and a set of resources that supply the shader with input data. This API gets a list of the resources that are bound as inputs to the shader.

Requirements

Target Platform	Windows
Header	d3d10shader.h

See also

ID3D10ShaderReflection Interface

ID3D10ShaderReflection1 interface (d3d10_1shader.h)

Article • 07/27/20222 minutes to read

A shader-reflection interface accesses shader information.

Inheritance

The **ID3D10ShaderReflection1** interface inherits from ID3D10ShaderReflection. **ID3D10ShaderReflection1** also has these types of members:

Methods

The **ID3D10ShaderReflection1** interface has these methods.

ID3D10ShaderReflection1::GetBitwiseInstructionCount Gets the number of bitwise instructions. (ID3D10ShaderReflection1.GetBitwiseInstructionCount)
ID3D10ShaderReflection1::GetConversionInstructionCount Gets the number of conversion instructions used in a shader.
ID3D10ShaderReflection1::GetGSInputPrimitive Gets the geometry-shader input-primitive description. (ID3D10ShaderReflection1.GetGSInputPrimitive)
ID3D10ShaderReflection1::GetMovcInstructionCount Gets the number of Movc instructions. (ID3D10ShaderReflection1.GetMovcInstructionCount)
ID3D10ShaderReflection1::GetMovInstructionCount Gets the number of Mov instructions. (ID3D10ShaderReflection1.GetMovInstructionCount)
ID3D10ShaderReflection1::GetResourceBindingDescByName Gets a resource binding description by name.

ID3D10ShaderReflection1::GetVariableByName Gets a variable by name. (ID3D10ShaderReflection1.GetVariableByName)	
ID3D10ShaderReflection1::IsLevel9Shader Indicates whether a shader was compiled in Direct3D 10 on Direct3D 9 mode.	
ID3D10ShaderReflection1::IsSampleFrequencyShader Indicates whether a pixel shader is intended to run a pixel frequency or sample frequency.	

Remarks

This interface requires Windows Vista Service Pack 1.

Requirements

Target Platform	Windows
Header	d3d10_1shader.h

See also

ID3D10ShaderReflection

Shader Interfaces

ID3D10ShaderReflection1::GetBitwiseInstructionCount method (d3d10_1shader.h)

Article • 07/27/20222 minutes to read

Gets the number of bitwise instructions.

Syntax

```
C++

HRESULT GetBitwiseInstructionCount(
  [in] UINT *pCount
);
```

Parameters

`[in] pCount`

Type: **UINT***

A pointer to the number of bitwise instructions.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Remarks

This method requires Windows Vista Service Pack 1.

Requirements

Target Platform	Windows
Header	d3d10_1shader.h

See also

ID3D10ShaderReflection1 Interface

ID3D10ShaderReflection1::GetConversionInstructionCount method (d3d10_1shader.h)

Article • 10/13/20212 minutes to read

Gets the number of conversion instructions used in a shader.

Syntax

```
C++

HRESULT GetConversionInstructionCount(
  [in] UINT *pCount
);
```

Parameters

`[in] pCount`

Type: UINT*

A pointer to a UINT that will contain the conversion instruction count when the method returns.

Return value

Type: HRESULT

Returns one of the following Direct3D 10 Return Codes.

Remarks

This method requires Windows Vista Service Pack 1.

Requirements

Target Platform	Windows
Header	d3d10_1shader.h

See also

ID3D10ShaderReflection1 Interface

ID3D10ShaderReflection1::GetGSInputPrimitive method (d3d10_1shader.h)

Article07/27/2022

Gets the geometry-shader input-primitive description.

Syntax

```
C++

HRESULT GetGSInputPrimitive(
  [in] D3D10_PRIMITIVE *pPrim
);
```

Parameters

`[in] pPrim`

Type: **D3D10_PRIMITIVE***

A pointer to the input-primitive type (see D3D10_PRIMITIVE).

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Remarks

This method requires Windows Vista Service Pack 1.

Requirements

Target Platform	Windows
Header	d3d10_1shader.h

See also

ID3D10ShaderReflection1 Interface

ID3D10ShaderReflection1::GetMovInstructionCount method (d3d10_1shader.h)

Article • 07/27/20222 minutes to read

Gets the number of Mov instructions.

Syntax

```
C++

HRESULT GetMovInstructionCount(
  [in] UINT *pCount
);
```

Parameters

`[in] pCount`

Type: **UINT***

A pointer to the number of Mov instructions.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Remarks

This method requires Windows Vista Service Pack 1.

Requirements

Target Platform	Windows
Header	d3d10_1shader.h

See also

ID3D10ShaderReflection1 Interface

ID3D10ShaderReflection1::GetMovcInstructionCount method (d3d10_1shader.h)

Article • 07/27/20222 minutes to read

Gets the number of Movc instructions.

Syntax

```
C++

HRESULT GetMovcInstructionCount(
  [in] UINT *pCount
);
```

Parameters

`[in] pCount`

Type: **UINT***

A pointer to the number of Movc instructions.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Remarks

This method requires Windows Vista Service Pack 1.

Requirements

Target Platform	Windows
Header	d3d10_1shader.h

See also

ID3D10ShaderReflection1 Interface

ID3D10ShaderReflection1::GetResourceBindingDescByName method (d3d10_1shader.h)

Article • 10/13/20212 minutes to read

Gets a resource binding description by name.

Syntax

```
C++

HRESULT GetResourceBindingDescByName(
  [in] LPCSTR                       Name,
       D3D10_SHADER_INPUT_BIND_DESC *pDesc
);
```

Parameters

`[in] Name`

Type: **LPCSTR***

A pointer to a string containing the variable name.

`pDesc`

Type: **D3D10_SHADER_INPUT_BIND_DESC***

Pointer to a D3D10_SHADER_INPUT_BIND_DESC structure that will be populated with resource binding information.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Remarks

This method requires Windows Vista Service Pack 1.

Requirements

Target Platform	Windows
Header	d3d10_1shader.h

See also

ID3D10ShaderReflection1 Interface

ID3D10ShaderReflection1::GetVariableByName method (d3d10_1shader.h)

Article • 07/27/20222 minutes to read

Gets a variable by name.

Syntax

```
C++

ID3D10ShaderReflectionVariable * GetVariableByName(
  [in] LPCSTR Name
);
```

Parameters

`[in] Name`

Type: **LPCSTR***

A pointer to a string containing the variable name.

Return value

Type: **ID3D10ShaderReflectionVariable***

Returns a ID3D10ShaderReflectionVariable Interface interface.

Remarks

This method requires Windows Vista Service Pack 1.

Requirements

Target Platform	Windows
Header	d3d10_1shader.h

See also

ID3D10ShaderReflection1 Interface

ID3D10ShaderReflection1::IsLevel9Shader method (d3d10_1shader.h)

Article • 10/13/20212 minutes to read

Indicates whether a shader was compiled in Direct3D 10 on Direct3D 9 mode.

Syntax

```
C++

HRESULT IsLevel9Shader(
  [in] BOOL *pbLevel9Shader
);
```

Parameters

`[in] pbLevel9Shader`

Type: **BOOL***

Pointer to a BOOL variable that will be set true if the shader was compiled in Direct3D 10 on Direct3D 9 mode; otherwise false.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Remarks

This method requires Windows Vista Service Pack 1.

Requirements

Target Platform	Windows

Header	d3d10_1shader.h

See also

ID3D10ShaderReflection1 Interface

ID3D10ShaderReflection1::IsSampleFrequencyShader method (d3d10_1shader.h)

Article • 04/02/20212 minutes to read

Indicates whether a pixel shader is intended to run a pixel frequency or sample frequency.

Syntax

```
C++

HRESULT IsSampleFrequencyShader(
  BOOL *pbSampleFrequency
);
```

Parameters

`pbSampleFrequency`

Type: **BOOL***

A pointer to a BOOL variable that will be set to true if the shader is intended to run at sample frequency; false otherwise.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Remarks

This method requires Windows Vista Service Pack 1.

Requirements

Target Platform	Windows
Header	d3d10_1shader.h

See also

ID3D10ShaderReflection1 Interface

ID3D10ShaderReflectionConstantBuffer interface (d3d10shader.h)

Article08/19/2022

This shader-reflection interface provides access to a constant buffer.

Methods

The **ID3D10ShaderReflectionConstantBuffer** interface has these methods.

ID3D10ShaderReflectionConstantBuffer::GetDesc Get a constant-buffer description. (ID3D10ShaderReflectionConstantBuffer.GetDesc)
ID3D10ShaderReflectionConstantBuffer::GetVariableByIndex The ID3D10ShaderReflectionConstantBuffer::GetVariableByIndex (d3d10shader.h) method gets a shader-reflection variable by index.
ID3D10ShaderReflectionConstantBuffer::GetVariableByName Get a shader-reflection variable by name. (ID3D10ShaderReflectionConstantBuffer.GetVariableByName)

Remarks

To create a constant-buffer interface, call ID3D10ShaderReflection::GetConstantBufferByIndex or ID3D10ShaderReflection::GetConstantBufferByName. This is not a COM interface; therefore, you do not need to worry about reference counts or releasing the interface when you are done with it.

Requirements

Target Platform	Windows
Header	d3d10shader.h

See also

Shader Interfaces

ID3D10ShaderReflectionConstantBuffer::GetDesc method (d3d10shader.h)

Article • 07/27/20222 minutes to read

Get a constant-buffer description.

Syntax

C++

```
HRESULT GetDesc(
  [in] D3D10_SHADER_BUFFER_DESC *pDesc
);
```

Parameters

`[in] pDesc`

Type: **D3D10_SHADER_BUFFER_DESC***

A pointer to a shader-buffer description (see D3D10_SHADER_BUFFER_DESC.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10shader.h

See also

ID3D10ShaderReflectionConstantBuffer Interface

ID3D10ShaderReflectionConstantBuffer::GetVariableByIndex method (d3d10shader.h)

Article • 08/19/20222 minutes to read

Get a shader-reflection variable by index.

Syntax

```
C++

ID3D10ShaderReflectionVariable * GetVariableByIndex(
  [in] UINT Index
);
```

Parameters

`[in] Index`

Type: **UINT**

Zero-based index.

Return value

Type: **ID3D10ShaderReflectionVariable***

A pointer to a shader-reflection variable interface (see ID3D10ShaderReflectionVariable Interface).

Requirements

Target Platform	Windows
Header	d3d10shader.h

See also

ID3D10ShaderReflectionConstantBuffer Interface

ID3D10ShaderReflectionConstantBuffer::GetVariableByName method (d3d10shader.h)

Article • 07/27/20222 minutes to read

Get a shader-reflection variable by name.

Syntax

```
C++

ID3D10ShaderReflectionVariable * GetVariableByName(
  [in] LPCSTR Name
);
```

Parameters

`[in] Name`

Type: **LPCSTR**

Variable name.

Return value

Type: **ID3D10ShaderReflectionVariable***

A pointer to a shader-reflection variable interface (see ID3D10ShaderReflectionVariable Interface).

Requirements

Target Platform	Windows
Header	d3d10shader.h

See also

ID3D10ShaderReflectionConstantBuffer Interface

ID3D10ShaderReflectionType interface (d3d10shader.h)

Article08/19/2022

This shader-reflection interface provides access to variable type.

Methods

The **ID3D10ShaderReflectionType** interface has these methods.

ID3D10ShaderReflectionType::GetDesc Get the description of a shader-reflection-variable type. (ID3D10ShaderReflectionType.GetDesc)
ID3D10ShaderReflectionType::GetMemberTypeByIndex The ID3D10ShaderReflectionType::GetMemberTypeByIndex (d3d10shader.h) method gets a shader-reflection-variable type by index.
ID3D10ShaderReflectionType::GetMemberTypeByName Get a shader-reflection-variable type by name. (ID3D10ShaderReflectionType.GetMemberTypeByName)
ID3D10ShaderReflectionType::GetMemberTypeName Get a shader-reflection-variable type. (ID3D10ShaderReflectionType.GetMemberTypeName)

Remarks

The get a shader-reflection-type interface, call ID3D10ShaderReflectionVariable::GetType.

Requirements

Target Platform	Windows
Header	d3d10shader.h

See also

Shader Interfaces

ID3D10ShaderReflectionType::GetDesc method (d3d10shader.h)

Article07/27/2022

Get the description of a shader-reflection-variable type.

Syntax

```
C++

HRESULT GetDesc(
  [in] D3D10_SHADER_TYPE_DESC *pDesc
);
```

Parameters

`[in] pDesc`

Type: **D3D10_SHADER_TYPE_DESC***

A pointer to a shader-type description (see D3D10_SHADER_TYPE_DESC).

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10shader.h

See also

ID3D10ShaderReflectionType Interface

ID3D10ShaderReflectionType::GetMemberTypeName method (d3d10shader.h)

Article • 07/27/20222 minutes to read

Retrieves a shader-reflection-variable name given the index to that member of the struct type.

Syntax

```
C++

LPCSTR GetMemberTypeName(
  [in] UINT Index
);
```

Parameters

`[in] Index`

Type: **UINT**

A zero-based index to a member of the struct type.

Return value

Type: **LPCSTR**

The member name in the form of a stringified value of the D3D_SHADER_VARIABLE_TYPE enumeration.

Requirements

Target Platform	Windows
Header	d3d10shader.h

See also

- ID3D10ShaderReflectionType interface
- D3D_SHADER_VARIABLE_TYPE enumeration

ID3D10ShaderReflectionType::GetMemberTypeByIndex method (d3d10shader.h)

Article • 08/19/20222 minutes to read

Get a shader-reflection-variable type by index.

Syntax

```
C++

ID3D10ShaderReflectionType * GetMemberTypeByIndex(
  [in] UINT Index
);
```

Parameters

`[in] Index`

Type: **UINT**

Zero-based index.

Return value

Type: **ID3D10ShaderReflectionType***

A pointer to a ID3D10ShaderReflectionType Interface.

Requirements

Target Platform	Windows
Header	d3d10shader.h

See also

ID3D10ShaderReflectionType Interface

ID3D10ShaderReflectionType::GetMemberTypeByName method (d3d10shader.h)

Article • 07/27/20222 minutes to read

Get a shader-reflection-variable type by name.

Syntax

```
C++

ID3D10ShaderReflectionType * GetMemberTypeByName(
  [in] LPCSTR Name
);
```

Parameters

`[in] Name`

Type: **LPCSTR**

Member name.

Return value

Type: **ID3D10ShaderReflectionType***

A pointer to a ID3D10ShaderReflectionType Interface.

Requirements

Target Platform	Windows
Header	d3d10shader.h

See also

ID3D10ShaderReflectionType Interface

ID3D10ShaderReflectionVariable interface (d3d10shader.h)

Article • 07/27/20222 minutes to read

This shader-reflection interface provides access to a variable.

Methods

The **ID3D10ShaderReflectionVariable** interface has these methods.

ID3D10ShaderReflectionVariable::GetDesc Get a shader-variable description. (ID3D10ShaderReflectionVariable.GetDesc)
ID3D10ShaderReflectionVariable::GetType Get a shader-variable type. (ID3D10ShaderReflectionVariable.GetType)

Remarks

To get a shader-reflection-variable interface, call a method like ID3D10ShaderReflectionConstantBuffer::GetVariableByIndex.

Requirements

Target Platform	Windows
Header	d3d10shader.h

See also

Shader Interfaces

ID3D10ShaderReflectionVariable::GetDesc method (d3d10shader.h)

Article • 07/27/20222 minutes to read

Get a shader-variable description.

Syntax

```
C++

HRESULT GetDesc(
  [in] D3D10_SHADER_VARIABLE_DESC *pDesc
);
```

Parameters

`[in] pDesc`

Type: **D3D10_SHADER_VARIABLE_DESC***

A pointer to a shader-variable description (see D3D10_SHADER_VARIABLE_DESC).

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10shader.h

See also

ID3D10ShaderReflectionVariable Interface

ID3D10ShaderReflectionVariable::GetType method (d3d10shader.h)

Article • 07/27/20222 minutes to read

Get a shader-variable type.

Syntax

```
C++

ID3D10ShaderReflectionType * GetType();
```

Return value

Type: **ID3D10ShaderReflectionType***

A pointer to a ID3D10ShaderReflectionType Interface.

Requirements

Target Platform	Windows
Header	d3d10shader.h

See also

ID3D10ShaderReflectionVariable Interface

ID3D10VertexShader interface (d3d10.h)

Article • 02/16/20232 minutes to read

A vertex-shader interface manages an executable program (a vertex shader) that controls the vertex-shader stage.

Inheritance

The **ID3D10VertexShader** interface inherits from the ID3D10DeviceChild interface.

Remarks

The vertex-shader interface has no methods; use HLSL to implement your shader functionality. All shaders in Direct3D 10 are implemented from a common set of features referred to as the common shader core.

To create a vertex shader interface, call ID3D10Device::CreateVertexShader. Before using a vertex shader you must bind it to the device by calling ID3D10Device::VSSetShader.

This interface is defined in D3D10.h.

Requirements

Target Platform	Windows
Header	d3d10.h

See also

ID3D10DeviceChild

Shader Interfaces

Shader Functions (Direct3D 10 Graphics)

Article • 01/06/2021 • 2 minutes to read

This section contains information about the following shader functions:

Functions	Description
D3D10CompileShader	Compile a shader.
D3D10DisassembleShader	Deprecated. See D3DX10DisassembleShader.
D3D10GetGeometryShaderProfile	Get the geometry-shader profile best suited to a given device.
D3D10GetInputAndOutputSignatureBlob	Get a buffer that contains shader signatures.
D3D10GetInputSignatureBlob	Get a buffer that contains shader input signatures.
D3D10GetOutputSignatureBlob	Get a buffer that contains shader output signatures.
D3D10GetPixelShaderProfile	Get the pixel-shader profile best suited to a given device.
D3D10GetShaderDebugInfo	Get shader debug info from a compiled shader.
D3D10GetVertexShaderProfile	Get the vertex-shader profile best suited to a given device.
D3D10PreprocessShader	Generate a text string that contains the shader token stream.
D3D10ReflectShader	Deprecated. See D3DX10ReflectShader.

Related topics

Shader Reference

D3D10CompileShader function (d3d10shader.h)

Article10/13/2021

Compile an HLSL shader.

> **Note** Use **D3DX10CompileFromMemory** instead of this function.

Syntax

```C++
HRESULT D3D10CompileShader(
  [in]  LPCSTR                   pSrcData,
  [in]  SIZE_T                   SrcDataSize,
  [in]  LPCSTR                   pFileName,
  [in]  const D3D10_SHADER_MACRO *pDefines,
  [in]  LPD3D10INCLUDE           pInclude,
  [in]  LPCSTR                   pFunctionName,
  [in]  LPCSTR                   pProfile,
  [in]  UINT                     Flags,
  [out] ID3D10Blob               **ppShader,
  [out] ID3D10Blob               **ppErrorMsgs
);
```

Parameters

`[in] pSrcData`

Type: **LPCSTR**

Pointer to a string containing the shader source code.

`[in] SrcDataSize`

Type: **SIZE_T**

Size of pSrcData, in bytes.

`[in] pFileName`

Type: **LPCSTR**

The name of the file that contains the shader code.

`[in] pDefines`

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**.

`[in] pInclude`

Type: **LPD3D10INCLUDE***

Optional. Pointer to an ID3D10Include Interface interface for handling include files. Setting this to **NULL** will cause a compile error if a shader contains a #include.

`[in] pFunctionName`

Type: **LPCSTR**

Name of the shader-entry point function where shader execution begins.

`[in] pProfile`

Type: **LPCSTR**

A string that specifies the shader profile or shader model.

`[in] Flags`

Type: **UINT**

Shader compile options.

`[out] ppShader`

Type: **ID3D10Blob****

A pointer to an ID3D10Blob Interface that contains the compiled shader, as well as any embedded debug and symbol-table information.

`[out] ppErrorMsgs`

Type: **ID3D10Blob****

A pointer to an ID3D10Blob Interface that contains a listing of errors and warnings that occurred during compilation. These errors and warnings are identical to the debug output from a debugger.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Remarks

This function uses the version of the HLSL compiler released in the November 2006 DirectX SDK.

This function implements two ways to supply the input shader information. Either use *pSrcData* and *SrcDataLen* to specify a string that contains the shader HLSL code (and set *pFileName* to **NULL**) or use *pFileName* to specify the name of a shader or effect file (and set *pSrcData* to **NULL**).

To setup a programmable-pipeline stage, compile a shader and then bind the shader to the appropriate pipeline stage. For instance, here is an example compiling a geometry shader (see Compile a Geometry Shader).

This function, D3D10CompileShader, calls the version of the shader compiler that is shipped each time the operating system releases. A more up-to-date version of the shader compiler ships when the DirectX SDK ships, which can be accessed from D3DX by calling a version of the shader compiler entry-point function such as D3DX10CompileFromFile. It is preferable to use the D3DX entry-point functions to ensure the latest version of the shader compiler will be used if you will be redistributing the DirectX redistributable libraries.

Requirements

Target Platform	Windows
Header	d3d10shader.h
Library	D3D10.lib
DLL	D3D10.dll

See also

Shader Functions

D3D10DisassembleShader function (d3d10shader.h)

Article • 10/13/20212 minutes to read

This function -- which disassembles a compiled shader into a text string that contains assembly instructions and register assignments -- has been deprecated. Instead, use D3DDisassemble.

Syntax

```
C++

HRESULT D3D10DisassembleShader(
  [in]  const void *pShader,
  [in]  SIZE_T     BytecodeLength,
  [in]  BOOL       EnableColorCode,
  [in]  LPCSTR     pComments,
  [out] ID3D10Blob **ppDisassembly
);
```

Parameters

`[in] pShader`

Type: **const void***

A pointer to the compiled shader.

`[in] BytecodeLength`

Type: **SIZE_T**

The size of pShader.

`[in] EnableColorCode`

Type: **BOOL**

Include HTML tags in the output to color code the result.

`[in] pComments`

Type: **LPCSTR**

The comment string at the top of the shader that identifies the shader constants and variables.

`[out] ppDisassembly`

Type: **ID3D10Blob****

Address of a buffer which contains the disassembled shader.

Return value

Type: **HRESULT**

Return value

Requirements

Target Platform	Windows
Header	d3d10shader.h
Library	D3D10.lib
DLL	D3D10.dll

See also

Shader Functions

D3D10GetGeometryShaderProfile function (d3d10shader.h)

Article • 10/13/20212 minutes to read

Get the geometry shader profile best suited to a given device.

Syntax

```
C++

LPCSTR D3D10GetGeometryShaderProfile(
  [in] ID3D10Device *pDevice
);
```

Parameters

`[in] pDevice`

Type: **ID3D10Device***

Pointer to the device (see ID3D10Device Interface).

Return value

Type: **LPCSTR**

The shader profile.

Requirements

Target Platform	Windows
Header	d3d10shader.h
Library	D3D10.lib
DLL	D3D10.dll

See also

Shader Functions

D3D10GetInputAndOutputSignatureBlob function (d3d10shader.h)

Article • 10/13/20212 minutes to read

Get a buffer that contains shader signatures.

Syntax

C++

```
HRESULT D3D10GetInputAndOutputSignatureBlob(
  [in]  const void *pShaderBytecode,
  [in]  SIZE_T     BytecodeLength,
  [out] ID3D10Blob **ppSignatureBlob
);
```

Parameters

`[in] pShaderBytecode`

Type: **const void***

A pointer to the compiled shader. To get this pointer see Getting a Pointer to a Compiled Shader.

`[in] BytecodeLength`

Type: **SIZE_T**

The size of the shader bytecode in bytes.

`[out] ppSignatureBlob`

Type: **ID3D10Blob****

The address of a pointer to the buffer (see ID3D10Blob Interface).

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10shader.h
Library	D3D10.lib
DLL	D3D10.dll

See also

Core Functions

Shader Functions

D3D10GetInputSignatureBlob function (d3d10shader.h)

Article • 10/13/20212 minutes to read

Get a buffer that contains shader-input signatures.

Syntax

C++

```
HRESULT D3D10GetInputSignatureBlob(
  [in]  const void *pShaderBytecode,
  [in]  SIZE_T     BytecodeLength,
  [out] ID3D10Blob **ppSignatureBlob
);
```

Parameters

`[in] pShaderBytecode`

Type: **const void***

A pointer to the compiled shader. To get this pointer see Getting a Pointer to a Compiled Shader.

`[in] BytecodeLength`

Type: **SIZE_T**

The size of the shader bytecode in bytes.

`[out] ppSignatureBlob`

Type: **ID3D10Blob****

The address of a pointer to the buffer (see ID3D10Blob Interface).

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10shader.h
Library	D3D10.lib
DLL	D3D10.dll

See also

Core Functions

Shader Functions

D3D10GetOutputSignatureBlob function (d3d10shader.h)

Article • 10/13/20212 minutes to read

Get a buffer that contains shader-output signatures.

Syntax

C++

```
HRESULT D3D10GetOutputSignatureBlob(
  [in]  const void *pShaderBytecode,
  [in]  SIZE_T     BytecodeLength,
  [out] ID3D10Blob **ppSignatureBlob
);
```

Parameters

`[in] pShaderBytecode`

Type: **const void***

A pointer to the compiled shader. To get this pointer see Getting a Pointer to a Compiled Shader.

`[in] BytecodeLength`

Type: **SIZE_T**

The size of the shader bytecode in bytes.

`[out] ppSignatureBlob`

Type: **ID3D10Blob****

The address of a pointer to the buffer (see ID3D10Blob Interface).

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10shader.h
Library	D3D10.lib
DLL	D3D10.dll

See also

Core Functions

Shader Functions

D3D10GetPixelShaderProfile function (d3d10shader.h)

Article10/13/2021

Get the pixel shader profile best suited to a given device.

Syntax

```
C++

LPCSTR D3D10GetPixelShaderProfile(
  [in] ID3D10Device *pDevice
);
```

Parameters

`[in] pDevice`

Type: **ID3D10Device***

Pointer to the device (see ID3D10Device Interface).

Return value

Type: **LPCSTR**

The shader profile.

Requirements

Target Platform	Windows
Header	d3d10shader.h
Library	D3D10.lib
DLL	D3D10.dll

See also

Shader Functions

D3D10GetShaderDebugInfo function (d3d10shader.h)

Article04/02/2021

Get shader debug info. Debug info is generated by D3D10CompileShader and is embedded in the body of the shader.

Syntax

```
C++

HRESULT D3D10GetShaderDebugInfo(
  const void *pShaderBytecode,
  SIZE_T     BytecodeLength,
  ID3D10Blob **ppDebugInfo
);
```

Parameters

`pShaderBytecode`

Type: **const void***

A pointer to the compiled shader. To get this pointer see Getting a Pointer to a Compiled Shader.

`BytecodeLength`

Type: **SIZE_T**

Length of the shader bytecode buffer.

`ppDebugInfo`

Type: **ID3D10Blob****

Pointer to an ID3D10Blob Interface used to return debug info. For information about the layout of this buffer, see D3D10_SHADER_DEBUG_INFO.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10shader.h
Library	D3D10.lib
DLL	D3D10.dll

See also

Shader Functions

D3D10GetVertexShaderProfile function (d3d10shader.h)

Article10/13/2021

Get the vertex shader profile best suited to a given device.

Syntax

```
C++

LPCSTR D3D10GetVertexShaderProfile(
  [in] ID3D10Device *pDevice
);
```

Parameters

`[in] pDevice`

Type: **ID3D10Device***

Pointer to the device (see ID3D10Device Interface).

Return value

Type: **LPCSTR**

The shader profile.

Requirements

Target Platform	Windows
Header	d3d10shader.h
Library	D3D10.lib
DLL	D3D10.dll

See also

Shader Functions

D3D10PreprocessShader function (d3d10shader.h)

Article • 10/13/20212 minutes to read

Generate a shader-text string that contains the shader tokens that would be found in a compiled shader.

Syntax

```C++
HRESULT D3D10PreprocessShader(
  [in]  LPCSTR                   pSrcData,
  [in]  SIZE_T                   SrcDataSize,
  [in]  LPCSTR                   pFileName,
  [in]  const D3D10_SHADER_MACRO *pDefines,
  [in]  LPD3D10INCLUDE           pInclude,
  [out] ID3D10Blob               **ppShaderText,
  [out] ID3D10Blob               **ppErrorMsgs
);
```

Parameters

`[in] pSrcData`

Type: **LPCSTR**

Pointer to a string containing the shader source code.

`[in] SrcDataSize`

Type: **SIZE_T**

Size of pSrcData, in bytes.

`[in] pFileName`

Type: **LPCSTR**

The name of the file that contains the shader code.

`[in] pDefines`

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**.

`[in] pInclude`

Type: **LPD3D10INCLUDE**

Optional. Pointer to an ID3D10Include Interface interface for handling include files. Setting this to **NULL** will cause a compile error if a shader contains a #include.

`[out] ppShaderText`

Type: **ID3D10Blob****

A pointer to a buffer that receives a pointer to an ID3D10Blob Interface that contains a single string containing shader-tokens.

`[out] ppErrorMsgs`

Type: **ID3D10Blob****

A pointer to a buffer that receives a pointer to an ID3D10Blob Interface that contains a listing of errors and warnings that occurred during compilation. These errors and warnings are identical to the debug output from a debugger.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Remarks

Use this function to generate a shader-token stream, which is the compiled output of the shader compiler.

Requirements

Target Platform	Windows
Header	d3d10shader.h
Library	D3D10.lib
DLL	D3D10.dll

See also

Shader Functions

D3D10ReflectShader function (d3d10shader.h)

Article • 10/13/20212 minutes to read

This function -- which creates a shader-reflection object for retrieving information about a compiled shader -- has been deprecated. Instead, use D3DReflect.

Syntax

```
C++

HRESULT D3D10ReflectShader(
  [in]  const void             *pShaderBytecode,
  [in]  SIZE_T                 BytecodeLength,
  [out] ID3D10ShaderReflection **ppReflector
);
```

Parameters

`[in] pShaderBytecode`

Type: **const void***

A pointer to the compiled shader.

`[in] BytecodeLength`

Type: **SIZE_T**

Length of pShaderBytecode.

`[out] ppReflector`

Type: **ID3D10ShaderReflection****

Address of a reflection interface.

Return value

Type: **HRESULT**

Return value.

Requirements

Target Platform	Windows
Header	d3d10shader.h
Library	D3D10.lib
DLL	D3D10.dll

See also

Shader Functions

Shader Structures (Direct3D 10 Graphics)

Article • 01/06/2021 • 2 minutes to read

This section contains information about the following shader structures:

Structures	Description
D3D10_SHADER_BUFFER_DESC	Describes a shader-constant buffer or a shader-texture buffer.
D3D10_SHADER_DESC	Describes a shader.
D3D10_SHADER_DEBUG_FILE_INFO	Describes files included by a shader.
D3D10_SHADER_DEBUG_INFO	Describes the format of the ID3D10Blob Interface returned by D3D10GetShaderDebugInfo.
D3D10_SHADER_DEBUG_INPUT_INFO	Describes a shader input.
D3D10_SHADER_DEBUG_INST_INFO	Contains instruction data.
D3D10_SHADER_DEBUG_OUTPUTREG_INFO	Describes a shader output register.
D3D10_SHADER_DEBUG_OUTPUTVAR	Describes a shader output variable.
D3D10_SHADER_DEBUG_SCOPE_INFO	Contains scope data that maps variable names to debug variables.
D3D10_SHADER_DEBUG_SCOPEVAR_INFO	Describes a scope variable.
D3D10_SHADER_DEBUG_TOKEN_INFO	Gives the source location for a shader element.
D3D10_SHADER_DEBUG_VARTYPE	Distinguishes variables from functions in a scope.
D3D10_SHADER_DEBUG_VAR_INFO	Contains information on a variable.
D3D10_SHADER_INPUT_BIND_DESC	Describes how a shader resource is bound to a shader input.
D3D_SHADER_MACRO	Defines a shader macro.
D3D10_SHADER_TYPE_DESC	Describes a shader-variable type.
D3D10_SHADER_VARIABLE_DESC	Describes a shader variable.
D3D10_SIGNATURE_PARAMETER_DESC	Describes a shader signature.

Related topics

Shader Reference

D3D10_SHADER_BUFFER_DESC structure (d3d10shader.h)

Article07/27/2022

Describes a shader constant-buffer.

Syntax

C++

```
typedef struct _D3D10_SHADER_BUFFER_DESC {
  LPCSTR             Name;
  D3D10_CBUFFER_TYPE Type;
  UINT               Variables;
  UINT               Size;
  UINT               uFlags;
} D3D10_SHADER_BUFFER_DESC;
```

Members

`Name`

Type: **LPCSTR**

The name of the buffer.

`Type`

Type: **D3D10_CBUFFER_TYPE**

The intended use of the constant data. See D3D10_CBUFFER_TYPE.

`Variables`

Type: **UINT**

The number of unique variables.

`Size`

Type: **UINT**

Buffer size (in bytes).

`uFlags`

Type: **UINT**

Shader buffer properties. See D3D10_SHADER_CBUFFER_FLAGS.

Remarks

Constants are supplied to shaders in a shader-constant buffer. Get the description of a shader-constant-buffer by calling ID3D10ShaderReflectionConstantBuffer::GetDesc.

Requirements

Header	d3d10shader.h

See also

Shader Structures

D3D10_SHADER_DESC structure (d3d10shader.h)

Article • 07/27/20222 minutes to read

Describes a shader.

Syntax

C++

```
typedef struct _D3D10_SHADER_DESC {
  UINT                     Version;
  LPCSTR                   Creator;
  UINT                     Flags;
  UINT                     ConstantBuffers;
  UINT                     BoundResources;
  UINT                     InputParameters;
  UINT                     OutputParameters;
  UINT                     InstructionCount;
  UINT                     TempRegisterCount;
  UINT                     TempArrayCount;
  UINT                     DefCount;
  UINT                     DclCount;
  UINT                     TextureNormalInstructions;
  UINT                     TextureLoadInstructions;
  UINT                     TextureCompInstructions;
  UINT                     TextureBiasInstructions;
  UINT                     TextureGradientInstructions;
  UINT                     FloatInstructionCount;
  UINT                     IntInstructionCount;
  UINT                     UintInstructionCount;
  UINT                     StaticFlowControlCount;
  UINT                     DynamicFlowControlCount;
  UINT                     MacroInstructionCount;
  UINT                     ArrayInstructionCount;
  UINT                     CutInstructionCount;
  UINT                     EmitInstructionCount;
  D3D10_PRIMITIVE_TOPOLOGY GSOutputTopology;
  UINT                     GSMaxOutputVertexCount;
} D3D10_SHADER_DESC;
```

Members

`Version`

Type: UINT

Shader version.

`Creator`

Type: **LPCSTR**

The name of the originator of the shader.

`Flags`

Type: **UINT**

Shader compilation/parse flags.

`ConstantBuffers`

Type: **UINT**

The number of shader-constant buffers.

`BoundResources`

Type: **UINT**

The number of resource (textures and buffers) bound to a shader.

`InputParameters`

Type: **UINT**

The number of parameters in the input signature.

`OutputParameters`

Type: **UINT**

The number of parameters in the output signature.

`InstructionCount`

Type: **UINT**

The number of intermediate-language instructions in the compiled shader.

`TempRegisterCount`

Type: **UINT**

The number of temporary registers in the compiled shader.

`TempArrayCount`

Type: **UINT**

Number of temporary arrays used.

`DefCount`

Type: **UINT**

Number of constant defines.

`DclCount`

Type: **UINT**

Number of declarations (input + output).

`TextureNormalInstructions`

Type: **UINT**

Number of non-categorized texture instructions.

`TextureLoadInstructions`

Type: **UINT**

Number of texture load instructions

`TextureCompInstructions`

Type: **UINT**

Number of texture comparison instructions

`TextureBiasInstructions`

Type: **UINT**

Number of texture bias instructions

`TextureGradientInstructions`

Type: **UINT**

Number of texture gradient instructions.

`FloatInstructionCount`

Type: **UINT**

Number of floating point arithmetic instructions used.

`IntInstructionCount`

Type: **UINT**

Number of signed integer arithmetic instructions used.

`UintInstructionCount`

Type: **UINT**

Number of unsigned integer arithmetic instructions used.

`StaticFlowControlCount`

Type: **UINT**

Number of static flow control instructions used.

`DynamicFlowControlCount`

Type: **UINT**

Number of dynamic flow control instructions used.

`MacroInstructionCount`

Type: **UINT**

Number of macro instructions used.

`ArrayInstructionCount`

Type: **UINT**

Number of array instructions used.

`CutInstructionCount`

Type: **UINT**

Number of cut instructions used.

`EmitInstructionCount`

Type: **UINT**

Number of emit instructions used.

`GSOutputTopology`

Type: **D3D10_PRIMITIVE_TOPOLOGY**

Geometry shader output topology.

`GSMaxOutputVertexCount`

Type: **UINT**

Geometry shader maximum output vertex count.

Remarks

A shader is written in HLSL and compiled into an intermediate language by the HLSL compiler. The shader description returns information about the compiled shader. Get a shader description by calling ID3D10ShaderReflection::GetDesc.

Requirements

Header	d3d10shader.h

See also

Shader Structures

D3D10_SHADER_DEBUG_FILE_INFO structure (d3d10_1shader.h)

Article • 04/02/20212 minutes to read

Describes files included by a shader.

Syntax

C++

```
typedef struct _D3D10_SHADER_DEBUG_FILE_INFO {
  UINT FileName;
  UINT FileNameLen;
  UINT FileData;
  UINT FileLen;
} D3D10_SHADER_DEBUG_FILE_INFO;
```

Members

`FileName`

Type: **UINT**

Offset to the LPCSTR for the file name.

`FileNameLen`

Type: **UINT**

Length of the file name.

`FileData`

Type: **UINT**

Offset to the file data.

`FileLen`

Type: **UINT**

Length of the file.

Remarks

The **D3D10_SHADER_DEBUG_FILE_INFO** structure is used with the D3D10_SHADER_DEBUG_INFO structure.

Requirements

Header	d3d10_1shader.h (include D3D10Shader.h)

See also

Shader Structures

D3D10_SHADER_DEBUG_INFO structure (d3d10_1shader.h)

Article • 04/02/20212 minutes to read

Describes the format of the ID3D10Blob Interface returned by D3D10GetShaderDebugInfo.

Syntax

C++

```
typedef struct _D3D10_SHADER_DEBUG_INFO {
  UINT Size;
  UINT Creator;
  UINT EntrypointName;
  UINT ShaderTarget;
  UINT CompileFlags;
  UINT Files;
  UINT FileInfo;
  UINT Instructions;
  UINT InstructionInfo;
  UINT Variables;
  UINT VariableInfo;
  UINT InputVariables;
  UINT InputVariableInfo;
  UINT Tokens;
  UINT TokenInfo;
  UINT Scopes;
  UINT ScopeInfo;
  UINT ScopeVariables;
  UINT ScopeVariableInfo;
  UINT UintOffset;
  UINT StringOffset;
} D3D10_SHADER_DEBUG_INFO;
```

Members

`Size`

Type: **UINT**

Size of this structure.

`Creator`

Type: **UINT**

Offset to LPCSTR for compiler version.

`EntrypointName`

Type: **UINT**

Offset to LPCSTR for Entry point name.

`ShaderTarget`

Type: **UINT**

Offset to LPCSTR for shader target.

`CompileFlags`

Type: **UINT**

Flags used to compile.

`Files`

Type: **UINT**

Number of included files.

`FileInfo`

Type: **UINT**

Offset to array of D3D10_SHADER_DEBUG_FILE_INFO structures that has **Files** elements.

`Instructions`

Type: **UINT**

Number of instructions.

`InstructionInfo`

Type: **UINT**

Offset to array of D3D10_SHADER_DEBUG_INST_INFO structures that has **Instructions** elements.

`Variables`

Type: **UINT**

Number of variables.

`VariableInfo`

Type: **UINT**

Offset to array of D3D10_SHADER_DEBUG_VAR_INFO structures that has **Variables** elements.

`InputVariables`

Type: **UINT**

Number of variables to initialize before running.

`InputVariableInfo`

Type: **UINT**

Offset to array of D3D10_SHADER_DEBUG_INPUT_INFO structures that has **InputVariables** elements.

`Tokens`

Type: **UINT**

Number of tokens to initialize.

`TokenInfo`

Type: **UINT**

Offset to array of D3D10_SHADER_DEBUG_TOKEN_INFO structures that has **Tokens** elements.

`Scopes`

Type: **UINT**

Number of scopes.

`ScopeInfo`

Type: **UINT**

Offset to array of D3D10_SHADER_DEBUG_SCOPE_INFO structures that has **Scopes** elements.

`ScopeVariables`

Type: **UINT**

Number of variables declared.

`ScopeVariableInfo`

Type: **UINT**

Offset to array of D3D10_SHADER_DEBUG_SCOPEVAR_INFO structures that has **Scopes** elements.

`UintOffset`

Type: **UINT**

Offset to the UINT datastore, all UINT offsets are from this offset.

`StringOffset`

Type: **UINT**

Offset to the string datastore, all string offsets are from this offset.

Requirements

Header	d3d10_1shader.h (include D3D10Shader.h)

See also

Shader Structures

D3D10_SHADER_DEBUG_INPUT_INFO structure (d3d10_1shader.h)

Article • 04/02/20212 minutes to read

Describes a shader input.

Syntax

C++

```
typedef struct _D3D10_SHADER_DEBUG_INPUT_INFO {
  UINT                        Var;
  D3D10_SHADER_DEBUG_REGTYPE InitialRegisterSet;
  UINT                        InitialBank;
  UINT                        InitialRegister;
  UINT                        InitialComponent;
  UINT                        InitialValue;
} D3D10_SHADER_DEBUG_INPUT_INFO;
```

Members

`Var`

Type: **UINT**

Index into array of variables to initialize.

`InitialRegisterSet`

Type: **D3D10_SHADER_DEBUG_REGTYPE**

Must be D3D10_SHADER_DEBUG_REG_INPUT, D3D10_SHADER_DEBUG_REG_CBUFFER or D3D10_SHADER_DEBUG_REG_TBUFFER.

`InitialBank`

Type: **UINT**

Will contain a cbuffer or tbuffer slot, geometry shader input primitive number, identifying register for an indexable temp, or -1.

`InitialRegister`

Type: **UINT**

Register in register set. **InitialRegister** will be -1 if it is temporary.

`InitialComponent`

Type: **UINT**

Gives the component. **InitialComponent** will be -1 it is temporary.

`InitialValue`

Type: **UINT**

Initial value if the variable is a literal.

Remarks

The **D3D10_SHADER_DEBUG_INPUT_INFO** structure is used with the D3D10_SHADER_DEBUG_INFO structure.

Requirements

Header	d3d10_1shader.h (include D3D10Shader.h)

See also

Shader Structures

D3D10_SHADER_DEBUG_INST_INFO structure (d3d10_1shader.h)

Article • 09/01/20222 minutes to read

Contains instruction data.

Syntax

C++

```
typedef struct _D3D10_SHADER_DEBUG_INST_INFO {
  UINT                                Id;
  UINT                                Opcode;
  UINT                                uOutputs;
  D3D10_SHADER_DEBUG_OUTPUTREG_INFO pOutputs[2];
  UINT                                TokenId;
  UINT                                NestingLevel;
  UINT                                Scopes;
  UINT                                ScopeInfo;
  UINT                                AccessedVars;
  UINT                                AccessedVarsInfo;
} D3D10_SHADER_DEBUG_INST_INFO;
```

Members

`Id`

Type: **UINT**

Id of the instruction.

`Opcode`

Type: **UINT**

Type of instruction.

`uOutputs`

Type: **UINT**

Must be 0, 1 or 2.

`pOutputs[2]`

Type: **D3D10_SHADER_DEBUG_OUTPUTREG_INFO**

Array containing the outputs of the instruction.

`TokenId`

Type: **UINT**

Index into the list of tokens for this instruction's token.

`NestingLevel`

Type: **UINT**

Number of function calls deep this instruction is.

`Scopes`

Type: **UINT**

Number of scopes.

`ScopeInfo`

Type: **UINT**

Offset to an array of UINT values with **Scopes** elements.

`AccessedVars`

`AccessedVarsInfo`

Remarks

The **D3D10_SHADER_DEBUG_INST_INFO** structure is used with the D3D10_SHADER_DEBUG_INFO structure.

Requirements

Header	d3d10_1shader.h (include D3D10Shader.h)

See also

D3D10_SHADER_DEBUG_OUTPUTREG_INFO structure (d3d10_1shader.h)

Article • 09/01/20222 minutes to read

Describes a shader output register.

Syntax

C++

```
typedef struct _D3D10_SHADER_DEBUG_OUTPUTREG_INFO {
  D3D10_SHADER_DEBUG_REGTYPE   OutputRegisterSet;
  UINT                         OutputReg;
  UINT                         TempArrayReg;
  UINT                         OutputComponents[4];
  D3D10_SHADER_DEBUG_OUTPUTVAR OutputVars[4];
  UINT                         IndexReg;
  UINT                         IndexComp;
} D3D10_SHADER_DEBUG_OUTPUTREG_INFO;
```

Members

`OutputRegisterSet`

Type: **D3D10_SHADER_DEBUG_REGTYPE**

Must be D3D10_SHADER_DEBUG_REG_TEMP, D3D10_SHADER_DEBUG_REG_TEMPARRAY or D3D10_SHADER_DEBUG_REG_OUTPUT.

`OutputReg`

Type: **UINT**

A value of -1 indicates no output.

`TempArrayReg`

Type: **UINT**

If **OutputRegisterSet** is D3D10_SHADER_DEBUG_REG_TEMPARRAY this indicates which temp array.

`OutputComponents[4]`

Type: **UINT**

A value of -1 means the component is masked out.

`OutputVars[4]`

Type: **D3D10_SHADER_DEBUG_OUTPUTVAR**

Indicates which variable the instruction is writing per-component.

`IndexReg`

Type: **UINT**

Offset from OutputReg of the element being written to. Used when writing to an indexable temp array or an output.

`IndexComp`

Type: **UINT**

Offset from OutputReg of the element being written to. Used when writing to an indexable temp array or an output.

Requirements

Header	d3d10_1shader.h (include D3D10Shader.h)

See also

Shader Structures

D3D10_SHADER_DEBUG_OUTPUTVAR structure (d3d10_1shader.h)

Article • 04/02/20212 minutes to read

Describes a shader output variable.

Syntax

C++

```
typedef struct _D3D10_SHADER_DEBUG_OUTPUTVAR {
  UINT  Var;
  UINT  uValueMin;
  UINT  uValueMax;
  INT   iValueMin;
  INT   iValueMax;
  FLOAT fValueMin;
  FLOAT fValueMax;
  BOOL  bNaNPossible;
  BOOL  bInfPossible;
} D3D10_SHADER_DEBUG_OUTPUTVAR;
```

Members

`Var`

Type: **UINT**

The index variable being written to or if -1 it's not going to a variable.

`uValueMin`

Type: **UINT**

Minimum UINT value.

`uValueMax`

Type: **UINT**

Maximum UINT value.

`iValueMin`

Type: **INT**

Minimum INT value.

`iValueMax`

Type: **INT**

Maximum UINT value.

`fValueMin`

Type: **FLOAT**

Minimum FLOAT value.

`fValueMax`

Type: **FLOAT**

Maximum FLOAT value.

`bNaNPossible`

Type: **BOOL**

Indicates whether the output variable can evaluate to not a number.

`bInfPossible`

Type: **BOOL**

Indicates whether the output variable can evaluate to infinity.

Requirements

Header	d3d10_1shader.h (include D3D10Shader.h)

See also

Shader Structures

D3D10_SHADER_DEBUG_SCOPE_INFO structure (d3d10_1shader.h)

Article08/03/2021

Contains scope data that maps variable names to debug variables.

Syntax

C++

```
typedef struct _D3D10_SHADER_DEBUG_SCOPE_INFO {
  D3D10_SHADER_DEBUG_SCOPETYPE ScopeType;
  UINT                         Name;
  UINT                         uNameLen;
  UINT                         uVariables;
  UINT                         VariableData;
} D3D10_SHADER_DEBUG_SCOPE_INFO;
```

Members

`ScopeType`

Type: D3D10_SHADER_DEBUG_SCOPETYPE

Specifies the scope type.

`Name`

Type: UINT

Offset to the name of scope in the strings list.

`uNameLen`

Type: UINT

Length of the string pointed to by **Name**.

`uVariables`

Type: UINT

Number of variables.

`VariableData`

Type: **UINT**

Offset an array of UINT values with **uVariables** members containing the scope variable list.

Remarks

The **D3D10_SHADER_DEBUG_SCOPE_INFO** structure is used with the D3D10_SHADER_DEBUG_INFO structure.

Requirements

Header	d3d10_1shader.h (include D3D10Shader.h)

See also

Shader Structures

D3D10_SHADER_DEBUG_SCOPEVAR_INFO structure (d3d10_1shader.h)

Article • 04/02/20212 minutes to read

Describes a shader scope variable.

Syntax

C++

```
typedef struct _D3D10_SHADER_DEBUG_SCOPEVAR_INFO {
  UINT                        TokenId;
  D3D10_SHADER_DEBUG_VARTYPE  VarType;
  D3D10_SHADER_VARIABLE_CLASS Class;
  UINT                        Rows;
  UINT                        Columns;
  UINT                        StructMemberScope;
  UINT                        uArrayIndices;
  UINT                        ArrayElements;
  UINT                        ArrayStrides;
  UINT                        uVariables;
  UINT                        uFirstVariable;
} D3D10_SHADER_DEBUG_SCOPEVAR_INFO;
```

Members

`TokenId`

Type: **UINT**

Index into variable token.

`VarType`

Type: **D3D10_SHADER_DEBUG_VARTYPE**

Indicates whether this is a variable or function.

`Class`

Type: **D3D10_SHADER_VARIABLE_CLASS**

Indicates the variable class.

`Rows`

Type: **UINT**

Number of row for matrices.

`Columns`

Type: **UINT**

Number of columns for vectors or matrices.

`StructMemberScope`

Type: **UINT**

Gives a scope to look up struct members. This member will be -1 if **D3D10_SHADER_DEBUG_SCOPEVAR_INFO** does not refer to a struct.

`uArrayIndices`

Type: **UINT**

Number of array indices. For example a three dimensional array would have a value of 3 for **uArrayIndices**.

`ArrayElements`

Type: **UINT**

Offset to an array of UINT values **uArrayIndices** long. The array contains the maximum value for each index. For example an array a[3][2][1] would have the values {3,2,1} at the offset pointed to by **ArrayElements**.

`ArrayStrides`

Type: **UINT**

Offset to an array of UINT values **uArrayIndices** long. The array contains the stride for each array index. For example an array a[3][2][1] would have the values {2,1,1} at the offset pointed to by **ArrayStrides**.

`uVariables`

Type: **UINT**

Number of variables.

`uFirstVariable`

Type: **UINT**

Index of the first variable, later variables are offsets from this one.

Remarks

The **D3D10_SHADER_DEBUG_SCOPEVAR_INFO** structure is used with the D3D10_SHADER_DEBUG_INFO structure.

Requirements

Header	d3d10_1shader.h (include D3D10Shader.h)

See also

Shader Structures

D3D10_SHADER_DEBUG_TOKEN_INFO structure (d3d10_1shader.h)

Article • 04/02/20212 minutes to read

Gives the source location for a shader element.

Syntax

C++

```
typedef struct _D3D10_SHADER_DEBUG_TOKEN_INFO {
  UINT File;
  UINT Line;
  UINT Column;
  UINT TokenLength;
  UINT TokenId;
} D3D10_SHADER_DEBUG_TOKEN_INFO;
```

Members

`File`

Type: **UINT**

Offset into file list.

`Line`

Type: **UINT**

Line number.

`Column`

Type: **UINT**

Column number.

`TokenLength`

Type: **UINT**

Length of the token.

`TokenId`

Type: **UINT**

Offset to LPCSTR of length **TokenLength** in string datastore.

Requirements

Header	d3d10_1shader.h (include D3D10Shader.h)

See also

Shader Structures

D3D10_SHADER_DEBUG_VARTYPE enumeration (d3d10_1shader.h)

Article01/31/2022

Distinguishes variables from functions in a scope.

Syntax

C++

```
typedef enum _D3D10_SHADER_DEBUG_VARTYPE {
  D3D10_SHADER_DEBUG_VAR_VARIABLE,
  D3D10_SHADER_DEBUG_VAR_FUNCTION,
  D3D10_SHADER_DEBUG_VAR_FORCE_DWORD = 0x7fffffff
} D3D10_SHADER_DEBUG_VARTYPE;
```

Constants

`D3D10_SHADER_DEBUG_VAR_VARIABLE` Element is a variable.
`D3D10_SHADER_DEBUG_VAR_FUNCTION` Element is a function.
`D3D10_SHADER_DEBUG_VAR_FORCE_DWORD` Value: *0x7fffffff* 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.

Requirements

Header	d3d10_1shader.h (include D3D10Shader.h)

See also

Shader Structures

D3D10_SHADER_DEBUG_VAR_INFO structure (d3d10_1shader.h)

Article04/02/2021

Represents information about a shader source variable.

Syntax

C++

```
typedef struct _D3D10_SHADER_DEBUG_VAR_INFO {
  UINT                        TokenId;
  D3D10_SHADER_VARIABLE_TYPE Type;
  UINT                        Register;
  UINT                        Component;
  UINT                        ScopeVar;
  UINT                        ScopeVarOffset;
} D3D10_SHADER_DEBUG_VAR_INFO;
```

Members

`TokenId`

Type: **UINT**

Index into token list for declaring identifier.

`Type`

Type: **D3D10_SHADER_VARIABLE_TYPE**

The variable type. **Type** is only required for arrays.

`Register`

Type: **UINT**

Register the variable is stored in.

`Component`

Type: **UINT**

The original variable that declared this variable.

`ScopeVar`

Type: **UINT**

Offset into the scope variable array defined in D3D10_SHADER_DEBUG_INFO.

`ScopeVarOffset`

Type: **UINT**

This variable's offset in its **ScopeVar**.

Requirements

Header	d3d10_1shader.h (include D3D10Shader.h)

See also

Shader Structures

D3D10_SHADER_INPUT_BIND_DESC structure (d3d10shader.h)

Article07/27/2022

Describes how a shader resource is bound to a shader input.

Syntax

C++

```
typedef struct _D3D10_SHADER_INPUT_BIND_DESC {
  LPCSTR                     Name;
  D3D10_SHADER_INPUT_TYPE    Type;
  UINT                       BindPoint;
  UINT                       BindCount;
  UINT                       uFlags;
  D3D10_RESOURCE_RETURN_TYPE ReturnType;
  D3D10_SRV_DIMENSION        Dimension;
  UINT                       NumSamples;
} D3D10_SHADER_INPUT_BIND_DESC;
```

Members

`Name`

Type: **LPCSTR**

Name of the shader resource.

`Type`

Type: **D3D10_SHADER_INPUT_TYPE**

Identifies the type of data in the resource. See D3D10_SHADER_INPUT_TYPE.

`BindPoint`

Type: **UINT**

Starting bind point.

`BindCount`

Type: **UINT**

Number of contiguous bind points for arrays.

`uFlags`

Type: **UINT**

Shader input-parameter options. See D3D10_SHADER_INPUT_FLAGS.

`ReturnType`

Type: **D3D10_RESOURCE_RETURN_TYPE**

If the input is a texture, the return type. See D3D10_RESOURCE_RETURN_TYPE.

`Dimension`

Type: **D3D10_SRV_DIMENSION**

Identifies the amount of data in the resource. See D3D10_SRV_DIMENSION.

`NumSamples`

Type: **UINT**

The number of samples for a multisampled texture; otherwise 0.

Remarks

Get a shader-input-signature description by calling ID3D10ShaderReflection::GetResourceBindingDesc.

Requirements

Header	d3d10shader.h

See also

Shader Structures

D3D_SHADER_MACRO structure (d3dcommon.h)

Article04/02/2021

Defines a shader macro.

Syntax

C++

```
typedef struct _D3D_SHADER_MACRO {
  LPCSTR Name;
  LPCSTR Definition;
} D3D_SHADER_MACRO, *LPD3D_SHADER_MACRO;
```

Members

`Name`

The macro name.

`Definition`

The macro definition.

Remarks

You can use shader macros in your shaders. The **D3D_SHADER_MACRO** structure defines a single shader macro as shown in the following example:

```
D3D_SHADER_MACRO Shader_Macros[] = { "zero", "0", NULL, NULL };
```

The following shader or effect creation functions take an array of shader macros as an input parameter:

- D3D10CompileShader

- D3DX10CreateEffectFromFile
- D3DX10PreprocessShaderFromFile
- D3DX11CreateAsyncShaderPreprocessProcessor

Requirements

Header	d3dcommon.h

See also

Common Version Structures

D3D10_SHADER_TYPE_DESC structure (d3d10shader.h)

Article • 07/27/20222 minutes to read

Describes a shader-variable type.

Syntax

C++

```
typedef struct _D3D10_SHADER_TYPE_DESC {
  D3D10_SHADER_VARIABLE_CLASS Class;
  D3D10_SHADER_VARIABLE_TYPE  Type;
  UINT                        Rows;
  UINT                        Columns;
  UINT                        Elements;
  UINT                        Members;
  UINT                        Offset;
} D3D10_SHADER_TYPE_DESC;
```

Members

`Class`

Type: **D3D10_SHADER_VARIABLE_CLASS**

Identifies the variable class as one of scalar, vector, matrix or object. See D3D10_SHADER_VARIABLE_CLASS.

`Type`

Type: **D3D10_SHADER_VARIABLE_TYPE**

The variable type. See D3D10_SHADER_VARIABLE_TYPE.

`Rows`

Type: **UINT**

Number of rows in a matrix. Otherwise a numeric type returns 1, any other type returns 0.

`Columns`

Type: **UINT**

Number of columns in a matrix. Otherwise a numeric type returns 1, any other type returns 0.

`Elements`

Type: **UINT**

Number of elements in an array; otherwise 0.

`Members`

Type: **UINT**

Number of members in the structure; otherwise 0.

`Offset`

Type: **UINT**

Offset, in bytes, between the start of the parent structure and this variable.

Remarks

Get a shader-variable-type description by calling ID3D10ShaderReflectionType::GetDesc.

Requirements

Header	d3d10shader.h

See also

Shader Structures

D3D10_SHADER_VARIABLE_DESC structure (d3d10shader.h)

Article • 07/27/20222 minutes to read

Describes a shader variable.

Syntax

C++

```
typedef struct _D3D10_SHADER_VARIABLE_DESC {
  LPCSTR Name;
  UINT   StartOffset;
  UINT   Size;
  UINT   uFlags;
  LPVOID DefaultValue;
} D3D10_SHADER_VARIABLE_DESC;
```

Members

`Name`

Type: **LPCSTR**

The variable name.

`StartOffset`

Type: **UINT**

Offset from the start of the parent structure, to the beginning of the variable.

`Size`

Type: **UINT**

Size of the variable (in bytes).

`uFlags`

Type: **UINT**

Flags, which identify shader-variable properties (see D3D10_SHADER_VARIABLE_FLAGS).

`DefaultValue`

Type: **LPVOID**

The default value for initializing the variable.

Remarks

Get a shader-variable description using reflection, by calling ID3D10ShaderReflectionVariable::GetDesc.

Requirements

Header	d3d10shader.h

See also

Shader Structures

D3D10_SIGNATURE_PARAMETER_DESC structure (d3d10shader.h)

Article • 07/27/20222 minutes to read

Describes a shader signature.

Syntax

C++

```
typedef struct _D3D10_SIGNATURE_PARAMETER_DESC {
  LPCSTR                        SemanticName;
  UINT                          SemanticIndex;
  UINT                          Register;
  D3D10_NAME                    SystemValueType;
  D3D10_REGISTER_COMPONENT_TYPE ComponentType;
  BYTE                          Mask;
  BYTE                          ReadWriteMask;
} D3D10_SIGNATURE_PARAMETER_DESC;
```

Members

`SemanticName`

Type: **LPCSTR**

A per-parameter string that identifies how the data will be used. See Semantics (DirectX HLSL).

`SemanticIndex`

Type: **UINT**

Semantic index that modifies the semantic. Used to differentiate different parameters that use the same semantic.

`Register`

Type: **UINT**

The register that will contain this variable's data.

`SystemValueType`

Type: **D3D10_NAME**

A predefined string that determines the functionality of certain pipeline stages. See D3D10_NAME.

`ComponentType`

Type: **D3D10_REGISTER_COMPONENT_TYPE**

The per-component-data type that is stored in a register. See D3D10_REGISTER_COMPONENT_TYPE. Each register can store up to four-components of data.

`Mask`

Type: **BYTE**

Mask which indicates which components of a register are used.

`ReadWriteMask`

Type: **BYTE**

Mask which indicates whether a given component is never written (if the signature is an output signature) or always read (if the signature is an input signature). The mask is a combination of D3D10_REGISTER_COMPONENT_TYPE values.

Remarks

A shader can take n inputs and can produce m outputs. The order of the input (or output) parameters, their associated types, and any attached semantics make up the shader signature. Each shader has an input and an output signature.

When compiling a shader or an effect, some API calls validate shader signatures (such as D3D10CompileShader and D3D10CompileEffectFromMemory). That is, they compare the output signature of one shader (like a vertex shader) with the input signature of another shader (like a pixel shader). This ensures that a shader outputs data that is compatible with a downstream shader that is consuming that data. Compatible means that a shader signature is a exact-match subset of the preceding shader stage. Exact match means parameter types and semantics must exactly match. Subset means that a parameter that is not required by a downstream stage, does not need to include that parameter in its shader signature.

Get a shader-signature from a shader or an effect by calling APIs such as ID3D10ShaderReflection::GetInputParameterDesc or ID3D10EffectShaderVariable::GetInputSignatureElementDesc.

Requirements

Header	d3d10shader.h

See also

Shader Structures

Shader Enumerations (Direct3D 10 Graphics)

Article • 01/06/2021 • 2 minutes to read

This section contains information about the following shader enumerations:

Enumeration	Description
D3D10_CBUFFER_TYPE	These flags identify constant-buffer-data types.
D3D_INCLUDE_TYPE	These flags identify the location of an include file.
D3D_PRIMITIVE	These flags identify primitive types for a Direct3D 10.1 device.
D3D10_SHADER_DEBUG_REGTYPE	Shader register types.
D3D10_SHADER_DEBUG_SCOPETYPE	Scope types.
D3D_NAME	These flags identify shader parameters that use system-value semantics.
D3D_SHADER_CBUFFER_FLAGS	These flags identify constant-buffer properties.
D3D_SHADER_INPUT_FLAGS	These flags identify shader-input properties.
D3D_SHADER_INPUT_TYPE	These flags identify a shader-resource type.
D3D_SHADER_VARIABLE_CLASS	These flags identify the shader-variable class.
D3D_SHADER_VARIABLE_FLAGS	These flags identify a shader-variable property.
D3D_SHADER_VARIABLE_TYPE	These flags identify the shader-variable type.
D3D10_SRV_DIMENSION	These flags identify the type of resource that will be viewed for a Direct3D 10.0 device.
D3D10_SRV_DIMENSION1	These flags identify the type of resource that will be viewed for a Direct3D 10.1 device.

Related topics

Shader Reference

D3D_INCLUDE_TYPE enumeration (d3dcommon.h)

Article • 01/31/20222 minutes to read

Values that indicate the location of a shader #include file.

> **Note**
>
> For programming with Direct3D 10, this API has a type alias that begins `D3D10_` instead of `D3D_`. These Direct3D 10 type aliases are defined in `d3d10.h`, `d3d10misc.h`, and `d3d10shader.h`.

Syntax

C++

```
typedef enum _D3D_INCLUDE_TYPE {
  D3D_INCLUDE_LOCAL = 0,
  D3D_INCLUDE_SYSTEM,
  D3D10_INCLUDE_LOCAL,
  D3D10_INCLUDE_SYSTEM,
  D3D_INCLUDE_FORCE_DWORD = 0x7fffffff
} D3D_INCLUDE_TYPE;
```

Constants

`D3D_INCLUDE_LOCAL` Value: *0* The local directory.
`D3D_INCLUDE_SYSTEM` The system directory.
`D3D10_INCLUDE_LOCAL` The local directory.
`D3D10_INCLUDE_SYSTEM` The system directory.

`D3D_INCLUDE_FORCE_DWORD` Value: *0x7fffffff* 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. Do not use this value.

Remarks

You pass a **D3D_INCLUDE_TYPE**-typed value to the *IncludeType* parameter in a call to the ID3DInclude::Open method to indicate the location of the #include file.

Requirements

Header	d3dcommon.h

See also

Common Version Enumerations

D3D_INCLUDE_TYPE

ID3DInclude::Open

D3D_PRIMITIVE enumeration (d3dcommon.h)

Article • 01/31/20223 minutes to read

Indicates how the pipeline interprets geometry or hull shader input primitives.

> **Note**
>
> For programming with Direct3D 10, this API has a type alias that begins `D3D10_` instead of `D3D_`. These Direct3D 10 type aliases are defined in `d3d10.h`, `d3d10misc.h`, and `d3d10shader.h`.

Syntax

C++

```
typedef enum D3D_PRIMITIVE {
  D3D_PRIMITIVE_UNDEFINED = 0,
  D3D_PRIMITIVE_POINT = 1,
  D3D_PRIMITIVE_LINE = 2,
  D3D_PRIMITIVE_TRIANGLE = 3,
  D3D_PRIMITIVE_LINE_ADJ = 6,
  D3D_PRIMITIVE_TRIANGLE_ADJ = 7,
  D3D_PRIMITIVE_1_CONTROL_POINT_PATCH = 8,
  D3D_PRIMITIVE_2_CONTROL_POINT_PATCH = 9,
  D3D_PRIMITIVE_3_CONTROL_POINT_PATCH = 10,
  D3D_PRIMITIVE_4_CONTROL_POINT_PATCH = 11,
  D3D_PRIMITIVE_5_CONTROL_POINT_PATCH = 12,
  D3D_PRIMITIVE_6_CONTROL_POINT_PATCH = 13,
  D3D_PRIMITIVE_7_CONTROL_POINT_PATCH = 14,
  D3D_PRIMITIVE_8_CONTROL_POINT_PATCH = 15,
  D3D_PRIMITIVE_9_CONTROL_POINT_PATCH = 16,
  D3D_PRIMITIVE_10_CONTROL_POINT_PATCH = 17,
  D3D_PRIMITIVE_11_CONTROL_POINT_PATCH = 18,
  D3D_PRIMITIVE_12_CONTROL_POINT_PATCH = 19,
  D3D_PRIMITIVE_13_CONTROL_POINT_PATCH = 20,
  D3D_PRIMITIVE_14_CONTROL_POINT_PATCH = 21,
  D3D_PRIMITIVE_15_CONTROL_POINT_PATCH = 22,
  D3D_PRIMITIVE_16_CONTROL_POINT_PATCH = 23,
  D3D_PRIMITIVE_17_CONTROL_POINT_PATCH = 24,
  D3D_PRIMITIVE_18_CONTROL_POINT_PATCH = 25,
  D3D_PRIMITIVE_19_CONTROL_POINT_PATCH = 26,
  D3D_PRIMITIVE_20_CONTROL_POINT_PATCH = 27,
  D3D_PRIMITIVE_21_CONTROL_POINT_PATCH = 28,
  D3D_PRIMITIVE_22_CONTROL_POINT_PATCH = 29,
```

```
    D3D_PRIMITIVE_23_CONTROL_POINT_PATCH = 30,
    D3D_PRIMITIVE_24_CONTROL_POINT_PATCH = 31,
    D3D_PRIMITIVE_25_CONTROL_POINT_PATCH = 32,
    D3D_PRIMITIVE_26_CONTROL_POINT_PATCH = 33,
    D3D_PRIMITIVE_27_CONTROL_POINT_PATCH = 34,
    D3D_PRIMITIVE_28_CONTROL_POINT_PATCH = 35,
    D3D_PRIMITIVE_29_CONTROL_POINT_PATCH = 36,
    D3D_PRIMITIVE_30_CONTROL_POINT_PATCH = 37,
    D3D_PRIMITIVE_31_CONTROL_POINT_PATCH = 38,
    D3D_PRIMITIVE_32_CONTROL_POINT_PATCH = 39,
    D3D10_PRIMITIVE_UNDEFINED,
    D3D10_PRIMITIVE_POINT,
    D3D10_PRIMITIVE_LINE,
    D3D10_PRIMITIVE_TRIANGLE,
    D3D10_PRIMITIVE_LINE_ADJ,
    D3D10_PRIMITIVE_TRIANGLE_ADJ,
    D3D11_PRIMITIVE_UNDEFINED,
    D3D11_PRIMITIVE_POINT,
    D3D11_PRIMITIVE_LINE,
    D3D11_PRIMITIVE_TRIANGLE,
    D3D11_PRIMITIVE_LINE_ADJ,
    D3D11_PRIMITIVE_TRIANGLE_ADJ,
    D3D11_PRIMITIVE_1_CONTROL_POINT_PATCH,
    D3D11_PRIMITIVE_2_CONTROL_POINT_PATCH,
    D3D11_PRIMITIVE_3_CONTROL_POINT_PATCH,
    D3D11_PRIMITIVE_4_CONTROL_POINT_PATCH,
    D3D11_PRIMITIVE_5_CONTROL_POINT_PATCH,
    D3D11_PRIMITIVE_6_CONTROL_POINT_PATCH,
    D3D11_PRIMITIVE_7_CONTROL_POINT_PATCH,
    D3D11_PRIMITIVE_8_CONTROL_POINT_PATCH,
    D3D11_PRIMITIVE_9_CONTROL_POINT_PATCH,
    D3D11_PRIMITIVE_10_CONTROL_POINT_PATCH,
    D3D11_PRIMITIVE_11_CONTROL_POINT_PATCH,
    D3D11_PRIMITIVE_12_CONTROL_POINT_PATCH,
    D3D11_PRIMITIVE_13_CONTROL_POINT_PATCH,
    D3D11_PRIMITIVE_14_CONTROL_POINT_PATCH,
    D3D11_PRIMITIVE_15_CONTROL_POINT_PATCH,
    D3D11_PRIMITIVE_16_CONTROL_POINT_PATCH,
    D3D11_PRIMITIVE_17_CONTROL_POINT_PATCH,
    D3D11_PRIMITIVE_18_CONTROL_POINT_PATCH,
    D3D11_PRIMITIVE_19_CONTROL_POINT_PATCH,
    D3D11_PRIMITIVE_20_CONTROL_POINT_PATCH,
    D3D11_PRIMITIVE_21_CONTROL_POINT_PATCH,
    D3D11_PRIMITIVE_22_CONTROL_POINT_PATCH,
    D3D11_PRIMITIVE_23_CONTROL_POINT_PATCH,
    D3D11_PRIMITIVE_24_CONTROL_POINT_PATCH,
    D3D11_PRIMITIVE_25_CONTROL_POINT_PATCH,
    D3D11_PRIMITIVE_26_CONTROL_POINT_PATCH,
    D3D11_PRIMITIVE_27_CONTROL_POINT_PATCH,
    D3D11_PRIMITIVE_28_CONTROL_POINT_PATCH,
    D3D11_PRIMITIVE_29_CONTROL_POINT_PATCH,
    D3D11_PRIMITIVE_30_CONTROL_POINT_PATCH,
    D3D11_PRIMITIVE_31_CONTROL_POINT_PATCH,
    D3D11_PRIMITIVE_32_CONTROL_POINT_PATCH
  } ;
```

Constants

`D3D_PRIMITIVE_UNDEFINED` Value: *0*
`D3D_PRIMITIVE_POINT` Value: *1*
`D3D_PRIMITIVE_LINE` Value: *2*
`D3D_PRIMITIVE_TRIANGLE` Value: *3*
`D3D_PRIMITIVE_LINE_ADJ` Value: *6*
`D3D_PRIMITIVE_TRIANGLE_ADJ` Value: *7*
`D3D_PRIMITIVE_1_CONTROL_POINT_PATCH` Value: *8*
`D3D_PRIMITIVE_2_CONTROL_POINT_PATCH` Value: *9*
`D3D_PRIMITIVE_3_CONTROL_POINT_PATCH` Value: *10*
`D3D_PRIMITIVE_4_CONTROL_POINT_PATCH` Value: *11*
`D3D_PRIMITIVE_5_CONTROL_POINT_PATCH` Value: *12*
`D3D_PRIMITIVE_6_CONTROL_POINT_PATCH` Value: *13*
`D3D_PRIMITIVE_7_CONTROL_POINT_PATCH` Value: *14*
`D3D_PRIMITIVE_8_CONTROL_POINT_PATCH` Value: *15*
`D3D_PRIMITIVE_9_CONTROL_POINT_PATCH` Value: *16*

`D3D_PRIMITIVE_10_CONTROL_POINT_PATCH` Value: *17*
`D3D_PRIMITIVE_11_CONTROL_POINT_PATCH` Value: *18*
`D3D_PRIMITIVE_12_CONTROL_POINT_PATCH` Value: *19*
`D3D_PRIMITIVE_13_CONTROL_POINT_PATCH` Value: *20*
`D3D_PRIMITIVE_14_CONTROL_POINT_PATCH` Value: *21*
`D3D_PRIMITIVE_15_CONTROL_POINT_PATCH` Value: *22*
`D3D_PRIMITIVE_16_CONTROL_POINT_PATCH` Value: *23*
`D3D_PRIMITIVE_17_CONTROL_POINT_PATCH` Value: *24*
`D3D_PRIMITIVE_18_CONTROL_POINT_PATCH` Value: *25*
`D3D_PRIMITIVE_19_CONTROL_POINT_PATCH` Value: *26*
`D3D_PRIMITIVE_20_CONTROL_POINT_PATCH` Value: *27*
`D3D_PRIMITIVE_21_CONTROL_POINT_PATCH` Value: *28*
`D3D_PRIMITIVE_22_CONTROL_POINT_PATCH` Value: *29*
`D3D_PRIMITIVE_23_CONTROL_POINT_PATCH` Value: *30*
`D3D_PRIMITIVE_24_CONTROL_POINT_PATCH` Value: *31*
`D3D_PRIMITIVE_25_CONTROL_POINT_PATCH` Value: *32*

`D3D_PRIMITIVE_26_CONTROL_POINT_PATCH` Value: *33*
`D3D_PRIMITIVE_27_CONTROL_POINT_PATCH` Value: *34*
`D3D_PRIMITIVE_28_CONTROL_POINT_PATCH` Value: *35*
`D3D_PRIMITIVE_29_CONTROL_POINT_PATCH` Value: *36*
`D3D_PRIMITIVE_30_CONTROL_POINT_PATCH` Value: *37*
`D3D_PRIMITIVE_31_CONTROL_POINT_PATCH` Value: *38*
`D3D_PRIMITIVE_32_CONTROL_POINT_PATCH` Value: *39*
`D3D10_PRIMITIVE_UNDEFINED`
`D3D10_PRIMITIVE_POINT`
`D3D10_PRIMITIVE_LINE`
`D3D10_PRIMITIVE_TRIANGLE`
`D3D10_PRIMITIVE_LINE_ADJ`
`D3D10_PRIMITIVE_TRIANGLE_ADJ`
`D3D11_PRIMITIVE_UNDEFINED` The shader has not been initialized with an input primitive type.
`D3D11_PRIMITIVE_POINT` Interpret the input primitive as a point.
`D3D11_PRIMITIVE_LINE` Interpret the input primitive as a line.
`D3D11_PRIMITIVE_TRIANGLE` Interpret the input primitive as a triangle.
`D3D11_PRIMITIVE_LINE_ADJ` Interpret the input primitive as a line with adjacency data.

`D3D11_PRIMITIVE_TRIANGLE_ADJ` Interpret the input primitive as a triangle with adjacency data.
`D3D11_PRIMITIVE_1_CONTROL_POINT_PATCH` Interpret the input primitive as a control point patch.
`D3D11_PRIMITIVE_2_CONTROL_POINT_PATCH` Interpret the input primitive as a control point patch.
`D3D11_PRIMITIVE_3_CONTROL_POINT_PATCH` Interpret the input primitive as a control point patch.
`D3D11_PRIMITIVE_4_CONTROL_POINT_PATCH` Interpret the input primitive as a control point patch.
`D3D11_PRIMITIVE_5_CONTROL_POINT_PATCH` Interpret the input primitive as a control point patch.
`D3D11_PRIMITIVE_6_CONTROL_POINT_PATCH` Interpret the input primitive as a control point patch.
`D3D11_PRIMITIVE_7_CONTROL_POINT_PATCH` Interpret the input primitive as a control point patch.
`D3D11_PRIMITIVE_8_CONTROL_POINT_PATCH` Interpret the input primitive as a control point patch.
`D3D11_PRIMITIVE_9_CONTROL_POINT_PATCH` Interpret the input primitive as a control point patch.
`D3D11_PRIMITIVE_10_CONTROL_POINT_PATCH` Interpret the input primitive as a control point patch.
`D3D11_PRIMITIVE_11_CONTROL_POINT_PATCH` Interpret the input primitive as a control point patch.
`D3D11_PRIMITIVE_12_CONTROL_POINT_PATCH` Interpret the input primitive as a control point patch.
`D3D11_PRIMITIVE_13_CONTROL_POINT_PATCH` Interpret the input primitive as a control point patch.
`D3D11_PRIMITIVE_14_CONTROL_POINT_PATCH` Interpret the input primitive as a control point patch.
`D3D11_PRIMITIVE_15_CONTROL_POINT_PATCH` Interpret the input primitive as a control point patch.

`D3D11_PRIMITIVE_16_CONTROL_POINT_PATCH` Interpret the input primitive as a control point patch.
`D3D11_PRIMITIVE_17_CONTROL_POINT_PATCH` Interpret the input primitive as a control point patch.
`D3D11_PRIMITIVE_18_CONTROL_POINT_PATCH` Interpret the input primitive as a control point patch.
`D3D11_PRIMITIVE_19_CONTROL_POINT_PATCH` Interpret the input primitive as a control point patch.
`D3D11_PRIMITIVE_20_CONTROL_POINT_PATCH` Interpret the input primitive as a control point patch.
`D3D11_PRIMITIVE_21_CONTROL_POINT_PATCH` Interpret the input primitive as a control point patch.
`D3D11_PRIMITIVE_22_CONTROL_POINT_PATCH` Interpret the input primitive as a control point patch.
`D3D11_PRIMITIVE_23_CONTROL_POINT_PATCH` Interpret the input primitive as a control point patch.
`D3D11_PRIMITIVE_24_CONTROL_POINT_PATCH` Interpret the input primitive as a control point patch.
`D3D11_PRIMITIVE_25_CONTROL_POINT_PATCH` Interpret the input primitive as a control point patch.
`D3D11_PRIMITIVE_26_CONTROL_POINT_PATCH` Interpret the input primitive as a control point patch.
`D3D11_PRIMITIVE_27_CONTROL_POINT_PATCH` Interpret the input primitive as a control point patch.
`D3D11_PRIMITIVE_28_CONTROL_POINT_PATCH` Interpret the input primitive as a control point patch.
`D3D11_PRIMITIVE_29_CONTROL_POINT_PATCH` Interpret the input primitive as a control point patch.
`D3D11_PRIMITIVE_30_CONTROL_POINT_PATCH` Interpret the input primitive as a control point patch.
`D3D11_PRIMITIVE_31_CONTROL_POINT_PATCH` Interpret the input primitive as a control point patch.

`D3D11_PRIMITIVE_32_CONTROL_POINT_PATCH`	Interpret the input primitive as a control point patch.

Remarks

The ID3D11ShaderReflection::GetGSInputPrimitive method returns a **D3D11_PRIMITIVE**-typed value.

The **D3D11_PRIMITIVE** enumeration is type defined in the D3D11.h header file as a D3D_PRIMITIVE enumeration, which is fully defined in the D3DCommon.h header file.

```
typedef D3D_PRIMITIVE D3D11_PRIMITIVE;
```

Requirements

Header	d3dcommon.h

See also

Core Enumerations

D3D10_SHADER_DEBUG_REGTYPE enumeration (d3d10_1shader.h)

Article • 01/31/20222 minutes to read

Shader register types.

Syntax

C++

```
typedef enum _D3D10_SHADER_DEBUG_REGTYPE {
  D3D10_SHADER_DEBUG_REG_INPUT,
  D3D10_SHADER_DEBUG_REG_OUTPUT,
  D3D10_SHADER_DEBUG_REG_CBUFFER,
  D3D10_SHADER_DEBUG_REG_TBUFFER,
  D3D10_SHADER_DEBUG_REG_TEMP,
  D3D10_SHADER_DEBUG_REG_TEMPARRAY,
  D3D10_SHADER_DEBUG_REG_TEXTURE,
  D3D10_SHADER_DEBUG_REG_SAMPLER,
  D3D10_SHADER_DEBUG_REG_IMMEDIATECBUFFER,
  D3D10_SHADER_DEBUG_REG_LITERAL,
  D3D10_SHADER_DEBUG_REG_UNUSED,
  D3D11_SHADER_DEBUG_REG_INTERFACE_POINTERS,
  D3D11_SHADER_DEBUG_REG_UAV,
  D3D10_SHADER_DEBUG_REG_FORCE_DWORD = 0x7fffffff
} D3D10_SHADER_DEBUG_REGTYPE;
```

Constants

`D3D10_SHADER_DEBUG_REG_INPUT` Input register.
`D3D10_SHADER_DEBUG_REG_OUTPUT` Output register.
`D3D10_SHADER_DEBUG_REG_CBUFFER` Constant buffer register.
`D3D10_SHADER_DEBUG_REG_TBUFFER` Texture buffer register.

`D3D10_SHADER_DEBUG_REG_TEMP` Temporary register.
`D3D10_SHADER_DEBUG_REG_TEMPARRAY` Array of temporary registers.
`D3D10_SHADER_DEBUG_REG_TEXTURE` Texture register.
`D3D10_SHADER_DEBUG_REG_SAMPLER` Sampler register.
`D3D10_SHADER_DEBUG_REG_IMMEDIATECBUFFER` Immediate constant buffer register.
`D3D10_SHADER_DEBUG_REG_LITERAL` Literal register.
`D3D10_SHADER_DEBUG_REG_UNUSED` Unused register.
`D3D11_SHADER_DEBUG_REG_INTERFACE_POINTERS` Interface register.
`D3D11_SHADER_DEBUG_REG_UAV` Unordered Access View (UAV) register.
`D3D10_SHADER_DEBUG_REG_FORCE_DWORD` Value: *0x7fffffff* 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

The **D3D10_SHADER_DEBUG_REGTYPE** enumeration is used to specify register types in D3D10_SHADER_DEBUG_INPUT_INFO and D3D10_SHADER_DEBUG_OUTPUTREG_INFO.

Requirements

Header	d3d10_1shader.h

See also

Shader Enumerations

D3D10_SHADER_DEBUG_SCOPETYPE enumeration (d3d10_1shader.h)

Article01/31/2022

Scope types.

Syntax

C++

```
typedef enum _D3D10_SHADER_DEBUG_SCOPETYPE {
  D3D10_SHADER_DEBUG_SCOPE_GLOBAL,
  D3D10_SHADER_DEBUG_SCOPE_BLOCK,
  D3D10_SHADER_DEBUG_SCOPE_FORLOOP,
  D3D10_SHADER_DEBUG_SCOPE_STRUCT,
  D3D10_SHADER_DEBUG_SCOPE_FUNC_PARAMS,
  D3D10_SHADER_DEBUG_SCOPE_STATEBLOCK,
  D3D10_SHADER_DEBUG_SCOPE_NAMESPACE,
  D3D10_SHADER_DEBUG_SCOPE_ANNOTATION,
  D3D10_SHADER_DEBUG_SCOPE_FORCE_DWORD = 0x7fffffff
} D3D10_SHADER_DEBUG_SCOPETYPE;
```

Constants

`D3D10_SHADER_DEBUG_SCOPE_GLOBAL` Global scope.
`D3D10_SHADER_DEBUG_SCOPE_BLOCK` Block scope.
`D3D10_SHADER_DEBUG_SCOPE_FORLOOP` For loop scope.
`D3D10_SHADER_DEBUG_SCOPE_STRUCT` Structure scope.
`D3D10_SHADER_DEBUG_SCOPE_FUNC_PARAMS` Function parameter scope.
`D3D10_SHADER_DEBUG_SCOPE_STATEBLOCK` State block scope.

`D3D10_SHADER_DEBUG_SCOPE_NAMESPACE`	Name space scope.
`D3D10_SHADER_DEBUG_SCOPE_ANNOTATION`	Annotation scope.
`D3D10_SHADER_DEBUG_SCOPE_FORCE_DWORD`	Value: *0x7fffffff* 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

The **D3D10_SHADER_DEBUG_SCOPETYPE** enumeration is used to specify scope type in the D3D10_SHADER_DEBUG_SCOPE_INFO structure.

Requirements

Header	d3d10_1shader.h (include D3D10Shader.h)

See also

Shader Enumerations

D3D_NAME enumeration (d3dcommon.h)

Article • 01/31/20225 minutes to read

Values that identify shader parameters that use system-value semantics.

> **Note**
>
> For programming with Direct3D 10, this API has a type alias that begins `D3D10_` instead of `D3D_`. These Direct3D 10 type aliases are defined in `d3d10.h`, `d3d10misc.h`, and `d3d10shader.h`.

Syntax

C++

```
typedef enum D3D_NAME {
  D3D_NAME_UNDEFINED = 0,
  D3D_NAME_POSITION = 1,
  D3D_NAME_CLIP_DISTANCE = 2,
  D3D_NAME_CULL_DISTANCE = 3,
  D3D_NAME_RENDER_TARGET_ARRAY_INDEX = 4,
  D3D_NAME_VIEWPORT_ARRAY_INDEX = 5,
  D3D_NAME_VERTEX_ID = 6,
  D3D_NAME_PRIMITIVE_ID = 7,
  D3D_NAME_INSTANCE_ID = 8,
  D3D_NAME_IS_FRONT_FACE = 9,
  D3D_NAME_SAMPLE_INDEX = 10,
  D3D_NAME_FINAL_QUAD_EDGE_TESSFACTOR = 11,
  D3D_NAME_FINAL_QUAD_INSIDE_TESSFACTOR = 12,
  D3D_NAME_FINAL_TRI_EDGE_TESSFACTOR = 13,
  D3D_NAME_FINAL_TRI_INSIDE_TESSFACTOR = 14,
  D3D_NAME_FINAL_LINE_DETAIL_TESSFACTOR = 15,
  D3D_NAME_FINAL_LINE_DENSITY_TESSFACTOR = 16,
  D3D_NAME_BARYCENTRICS = 23,
  D3D_NAME_SHADINGRATE,
  D3D_NAME_CULLPRIMITIVE,
  D3D_NAME_TARGET = 64,
  D3D_NAME_DEPTH = 65,
  D3D_NAME_COVERAGE = 66,
  D3D_NAME_DEPTH_GREATER_EQUAL = 67,
  D3D_NAME_DEPTH_LESS_EQUAL = 68,
  D3D_NAME_STENCIL_REF = 69,
  D3D_NAME_INNER_COVERAGE = 70,
  D3D10_NAME_UNDEFINED,
```

```
  D3D10_NAME_POSITION,
  D3D10_NAME_CLIP_DISTANCE,
  D3D10_NAME_CULL_DISTANCE,
  D3D10_NAME_RENDER_TARGET_ARRAY_INDEX,
  D3D10_NAME_VIEWPORT_ARRAY_INDEX,
  D3D10_NAME_VERTEX_ID,
  D3D10_NAME_PRIMITIVE_ID,
  D3D10_NAME_INSTANCE_ID,
  D3D10_NAME_IS_FRONT_FACE,
  D3D10_NAME_SAMPLE_INDEX,
  D3D10_NAME_TARGET,
  D3D10_NAME_DEPTH,
  D3D10_NAME_COVERAGE,
  D3D11_NAME_FINAL_QUAD_EDGE_TESSFACTOR,
  D3D11_NAME_FINAL_QUAD_INSIDE_TESSFACTOR,
  D3D11_NAME_FINAL_TRI_EDGE_TESSFACTOR,
  D3D11_NAME_FINAL_TRI_INSIDE_TESSFACTOR,
  D3D11_NAME_FINAL_LINE_DETAIL_TESSFACTOR,
  D3D11_NAME_FINAL_LINE_DENSITY_TESSFACTOR,
  D3D11_NAME_DEPTH_GREATER_EQUAL,
  D3D11_NAME_DEPTH_LESS_EQUAL,
  D3D11_NAME_STENCIL_REF,
  D3D11_NAME_INNER_COVERAGE,
  D3D12_NAME_BARYCENTRICS,
  D3D12_NAME_SHADINGRATE,
  D3D12_NAME_CULLPRIMITIVE
} ;
```

Constants

`D3D_NAME_UNDEFINED` Value: *0* This parameter does not use a predefined system-value semantic.
`D3D_NAME_POSITION` Value: *1* This parameter contains position data.
`D3D_NAME_CLIP_DISTANCE` Value: *2* This parameter contains clip-distance data.
`D3D_NAME_CULL_DISTANCE` Value: *3* This parameter contains cull-distance data.

`D3D_NAME_RENDER_TARGET_ARRAY_INDEX` Value: *4* This parameter contains a render-target-array index.
`D3D_NAME_VIEWPORT_ARRAY_INDEX` Value: *5* This parameter contains a viewport-array index.
`D3D_NAME_VERTEX_ID` Value: *6* This parameter contains a vertex ID.
`D3D_NAME_PRIMITIVE_ID` Value: *7* This parameter contains a primitive ID.
`D3D_NAME_INSTANCE_ID` Value: *8* This parameter contains an instance ID.
`D3D_NAME_IS_FRONT_FACE` Value: *9* This parameter contains data that identifies whether or not the primitive faces the camera.
`D3D_NAME_SAMPLE_INDEX` Value: *10* This parameter contains a sampler-array index.
`D3D_NAME_FINAL_QUAD_EDGE_TESSFACTOR` Value: *11* This parameter contains one of four tessellation factors that correspond to the amount of parts that a quad patch is broken into along the given edge. This flag is used to tessellate a quad patch.
`D3D_NAME_FINAL_QUAD_INSIDE_TESSFACTOR` Value: *12* This parameter contains one of two tessellation factors that correspond to the amount of parts that a quad patch is broken into vertically and horizontally within the patch. This flag is used to tessellate a quad patch.
`D3D_NAME_FINAL_TRI_EDGE_TESSFACTOR` Value: *13* This parameter contains one of three tessellation factors that correspond to the amount of parts that a tri patch is broken into along the given edge. This flag is used to tessellate a tri patch.
`D3D_NAME_FINAL_TRI_INSIDE_TESSFACTOR` Value: *14* This parameter contains the tessellation factor that corresponds to the amount of parts that a tri patch is broken into within the patch. This flag is used to tessellate a tri patch.

`D3D_NAME_FINAL_LINE_DETAIL_TESSFACTOR` Value: *15* This parameter contains the tessellation factor that corresponds to the number of lines broken into within the patch. This flag is used to tessellate an isolines patch.
`D3D_NAME_FINAL_LINE_DENSITY_TESSFACTOR` Value: *16* This parameter contains the tessellation factor that corresponds to the number of lines that are created within the patch. This flag is used to tessellate an isolines patch.
`D3D_NAME_BARYCENTRICS` Value: *23* This parameter contains barycentric coordinate data.
`D3D_NAME_TARGET` Value: *64* This parameter contains render-target data.
`D3D_NAME_DEPTH` Value: *65* This parameter contains depth data.
`D3D_NAME_COVERAGE` Value: *66* This parameter contains alpha-coverage data.
`D3D_NAME_DEPTH_GREATER_EQUAL` Value: *67* This parameter signifies that the value is greater than or equal to a reference value. This flag is used to specify conservative depth for a pixel shader.
`D3D_NAME_DEPTH_LESS_EQUAL` Value: *68* This parameter signifies that the value is less than or equal to a reference value. This flag is used to specify conservative depth for a pixel shader.
`D3D_NAME_STENCIL_REF` Value: *69* This parameter contains a stencil reference. See Shader Specified Stencil Reference Value.
`D3D_NAME_INNER_COVERAGE` Value: *70* This parameter contains inner input coverage data. See Conservative Rasterization.
`D3D10_NAME_UNDEFINED` This parameter does not use a predefined system-value semantic.

`D3D10_NAME_POSITION` This parameter contains position data.
`D3D10_NAME_CLIP_DISTANCE` This parameter contains clip-distance data.
`D3D10_NAME_CULL_DISTANCE` This parameter contains cull-distance data.
`D3D10_NAME_RENDER_TARGET_ARRAY_INDEX` This parameter contains a render-target-array index.
`D3D10_NAME_VIEWPORT_ARRAY_INDEX` This parameter contains a viewport-array index.
`D3D10_NAME_VERTEX_ID` This parameter contains a vertex ID.
`D3D10_NAME_PRIMITIVE_ID` This parameter contains a primitive ID.
`D3D10_NAME_INSTANCE_ID` This parameter contains a instance ID.
`D3D10_NAME_IS_FRONT_FACE` This parameter contains data that identifies whether or not the primitive faces the camera.
`D3D10_NAME_SAMPLE_INDEX` This parameter contains a sampler-array index.
`D3D10_NAME_TARGET` This parameter contains render-target data.
`D3D10_NAME_DEPTH` This parameter contains depth data.
`D3D10_NAME_COVERAGE` This parameter contains alpha-coverage data.
`D3D11_NAME_FINAL_QUAD_EDGE_TESSFACTOR` This parameter contains one of four tessellation factors that correspond to the amount of parts that a quad patch is broken into along the given edge. This flag is used to tessellate a quad patch.
`D3D11_NAME_FINAL_QUAD_INSIDE_TESSFACTOR` This parameter contains one of two tessellation factors that correspond to the amount of parts that a quad patch is broken into vertically and horizontally within the patch. This flag is used to tessellate a quad patch.

`D3D11_NAME_FINAL_TRI_EDGE_TESSFACTOR`	This parameter contains one of three tessellation factors that correspond to the amount of parts that a tri patch is broken into along the given edge. This flag is used to tessellate a tri patch.
`D3D11_NAME_FINAL_TRI_INSIDE_TESSFACTOR`	This parameter contains the tessellation factor that corresponds to the amount of parts that a tri patch is broken into within the patch. This flag is used to tessellate a tri patch.
`D3D11_NAME_FINAL_LINE_DETAIL_TESSFACTOR`	This parameter contains the tessellation factor that corresponds to the amount of lines broken into within the patch. This flag is used to tessellate an isolines patch.
`D3D11_NAME_FINAL_LINE_DENSITY_TESSFACTOR`	This parameter contains the tessellation factor that corresponds to the amount of lines that are created within the patch. This flag is used to tessellate an isolines patch.
`D3D11_NAME_DEPTH_GREATER_EQUAL`	This parameter signifies that the value is greater than or equal to a reference value. This flag is used to specify conservative depth for a pixel shader.
`D3D11_NAME_DEPTH_LESS_EQUAL`	This parameter signifies that the value is less than or equal to a reference value. This flag is used to specify conservative depth for a pixel shader.
`D3D11_NAME_STENCIL_REF`	This parameter contains a stencil reference. See Shader Specified Stencil Reference Value.
`D3D11_NAME_INNER_COVERAGE`	This parameter contains inner input coverage data. See Conservative Rasterization.
`D3D12_NAME_BARYCENTRICS`	This parameter contains barycentric coordinate data.

Remarks

The **D3D_NAME** values identify shader parameters that have predefined system-value semantics. These values are used in a shader-signature description. For more information about shader-signature description, see D3D11_SIGNATURE_PARAMETER_DESC.

Requirements

Header	d3dcommon.h

See also

Common Version Enumerations

D3D11_SIGNATURE_PARAMETER_DESC

D3D_SHADER_CBUFFER_FLAGS enumeration (d3dcommon.h)

Article • 01/31/20222 minutes to read

Values that identify the intended use of a constant-data buffer.

> **Note**
>
> For programming with Direct3D 10, this API has a type alias that begins `D3D10_` instead of `D3D_`. These Direct3D 10 type aliases are defined in `d3d10.h`, `d3d10misc.h`, and `d3d10shader.h`.

Syntax

C++

```
typedef enum _D3D_SHADER_CBUFFER_FLAGS {
  D3D_CBF_USERPACKED = 1,
  D3D10_CBF_USERPACKED,
  D3D_CBF_FORCE_DWORD = 0x7fffffff
} D3D_SHADER_CBUFFER_FLAGS;
```

Constants

`D3D_CBF_USERPACKED` Value: *1* Bind the constant buffer to an input slot defined in HLSL code (instead of letting the compiler choose the input slot).
`D3D10_CBF_USERPACKED` Bind the constant buffer to an input slot defined in HLSL code (instead of letting the compiler choose the input slot).
`D3D_CBF_FORCE_DWORD` Value: *0x7fffffff* This value is not used by a programmer; it exists to force the enumeration to compile to 32 bits.

Remarks

D3D_SHADER_CBUFFER_FLAGS-typed values are specified in the **uFlags** member of the D3D11_SHADER_BUFFER_DESC structure.

Requirements

Header	d3dcommon.h

See also

Common Version Enumerations

D3D_SHADER_INPUT_FLAGS enumeration (d3dcommon.h)

Article • 01/31/20222 minutes to read

Values that identify shader-input options.

> **Note**
>
> For programming with Direct3D 10, this API has a type alias that begins `D3D10_` instead of `D3D_`. These Direct3D 10 type aliases are defined in `d3d10.h`, `d3d10misc.h`, and `d3d10shader.h`.

Syntax

C++

```
typedef enum _D3D_SHADER_INPUT_FLAGS {
  D3D_SIF_USERPACKED = 0x1,
  D3D_SIF_COMPARISON_SAMPLER = 0x2,
  D3D_SIF_TEXTURE_COMPONENT_0 = 0x4,
  D3D_SIF_TEXTURE_COMPONENT_1 = 0x8,
  D3D_SIF_TEXTURE_COMPONENTS = 0xc,
  D3D_SIF_UNUSED = 0x10,
  D3D10_SIF_USERPACKED,
  D3D10_SIF_COMPARISON_SAMPLER,
  D3D10_SIF_TEXTURE_COMPONENT_0,
  D3D10_SIF_TEXTURE_COMPONENT_1,
  D3D10_SIF_TEXTURE_COMPONENTS,
  D3D_SIF_FORCE_DWORD = 0x7fffffff
} D3D_SHADER_INPUT_FLAGS;
```

Constants

`D3D_SIF_USERPACKED` Value: *0x1* Assign a shader input to a register based on the register assignment in the HLSL code (instead of letting the compiler choose the register).

`D3D_SIF_COMPARISON_SAMPLER` Value: *0x2* Use a comparison sampler, which uses the SampleCmp (DirectX HLSL Texture Object) and SampleCmpLevelZero (DirectX HLSL Texture Object) sampling functions.
`D3D_SIF_TEXTURE_COMPONENT_0` Value: *0x4* A 2-bit value for encoding texture components.
`D3D_SIF_TEXTURE_COMPONENT_1` Value: *0x8* A 2-bit value for encoding texture components.
`D3D_SIF_TEXTURE_COMPONENTS` Value: *0xc* A 2-bit value for encoding texture components.
`D3D_SIF_UNUSED` Value: *0x10* This value is reserved.
`D3D10_SIF_USERPACKED` Assign a shader input to a register based on the register assignment in the HLSL code (instead of letting the compiler choose the register).
`D3D10_SIF_COMPARISON_SAMPLER` Use a comparison sampler, which uses the SampleCmp (DirectX HLSL Texture Object) and SampleCmpLevelZero (DirectX HLSL Texture Object) sampling functions.
`D3D10_SIF_TEXTURE_COMPONENT_0` A 2-bit value for encoding texture components.
`D3D10_SIF_TEXTURE_COMPONENT_1` A 2-bit value for encoding texture components.
`D3D10_SIF_TEXTURE_COMPONENTS` A 2-bit value for encoding texture components.
`D3D_SIF_FORCE_DWORD` Value: *0x7fffffff* Forces the enumeration to compile to 32 bits. This value is not used directly by titles.

Remarks

D3D_SHADER_INPUT_FLAGS-typed values are specified in the **uFlags** member of the D3D11_SHADER_INPUT_BIND_DESC structure.

Requirements

Header	d3dcommon.h

See also

Common Version Enumerations

D3D_SHADER_INPUT_TYPE enumeration (d3dcommon.h)

Article • 01/31/20222 minutes to read

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.

> **Note**
>
> For programming with Direct3D 10, this API has a type alias that begins `D3D10_` instead of `D3D_`. These Direct3D 10 type aliases are defined in `d3d10.h`, `d3d10misc.h`, and `d3d10shader.h`.

Syntax

C++

```
typedef enum _D3D_SHADER_INPUT_TYPE {
  D3D_SIT_CBUFFER = 0,
  D3D_SIT_TBUFFER,
  D3D_SIT_TEXTURE,
  D3D_SIT_SAMPLER,
  D3D_SIT_UAV_RWTYPED,
  D3D_SIT_STRUCTURED,
  D3D_SIT_UAV_RWSTRUCTURED,
  D3D_SIT_BYTEADDRESS,
  D3D_SIT_UAV_RWBYTEADDRESS,
  D3D_SIT_UAV_APPEND_STRUCTURED,
  D3D_SIT_UAV_CONSUME_STRUCTURED,
  D3D_SIT_UAV_RWSTRUCTURED_WITH_COUNTER,
  D3D_SIT_RTACCELERATIONSTRUCTURE,
  D3D_SIT_UAV_FEEDBACKTEXTURE,
  D3D10_SIT_CBUFFER,
  D3D10_SIT_TBUFFER,
  D3D10_SIT_TEXTURE,
  D3D10_SIT_SAMPLER,
  D3D11_SIT_UAV_RWTYPED,
  D3D11_SIT_STRUCTURED,
  D3D11_SIT_UAV_RWSTRUCTURED,
  D3D11_SIT_BYTEADDRESS,
  D3D11_SIT_UAV_RWBYTEADDRESS,
  D3D11_SIT_UAV_APPEND_STRUCTURED,
  D3D11_SIT_UAV_CONSUME_STRUCTURED,
  D3D11_SIT_UAV_RWSTRUCTURED_WITH_COUNTER
} D3D_SHADER_INPUT_TYPE;
```

Constants

`D3D_SIT_CBUFFER` Value: *0* The shader resource is a constant buffer.
`D3D_SIT_TBUFFER` The shader resource is a texture buffer.
`D3D_SIT_TEXTURE` The shader resource is a texture.
`D3D_SIT_SAMPLER` The shader resource is a sampler.
`D3D_SIT_UAV_RWTYPED` The shader resource is a read-and-write buffer.
`D3D_SIT_STRUCTURED` The shader resource is a structured buffer. For more information about structured buffer, see the **Remarks** section.
`D3D_SIT_UAV_RWSTRUCTURED` The shader resource is a read-and-write structured buffer.
`D3D_SIT_BYTEADDRESS` The shader resource is a byte-address buffer.
`D3D_SIT_UAV_RWBYTEADDRESS` The shader resource is a read-and-write byte-address buffer.
`D3D_SIT_UAV_APPEND_STRUCTURED` The shader resource is an append-structured buffer.
`D3D_SIT_UAV_CONSUME_STRUCTURED` The shader resource is a consume-structured buffer.
`D3D_SIT_UAV_RWSTRUCTURED_WITH_COUNTER` The shader resource is a read-and-write structured buffer that uses the built-in counter to append or consume.
`D3D10_SIT_CBUFFER` The shader resource is a constant buffer.

`D3D10_SIT_TBUFFER` The shader resource is a texture buffer.
`D3D10_SIT_TEXTURE` The shader resource is a texture.
`D3D10_SIT_SAMPLER` The shader resource is a sampler.
`D3D11_SIT_UAV_RWTYPED` The shader resource is a read-and-write buffer.
`D3D11_SIT_STRUCTURED` The shader resource is a structured buffer. For more information about structured buffer, see the **Remarks** section.
`D3D11_SIT_UAV_RWSTRUCTURED` The shader resource is a read-and-write structured buffer.
`D3D11_SIT_BYTEADDRESS` The shader resource is a byte-address buffer.
`D3D11_SIT_UAV_RWBYTEADDRESS` The shader resource is a read-and-write byte-address buffer.
`D3D11_SIT_UAV_APPEND_STRUCTURED` The shader resource is an append-structured buffer.
`D3D11_SIT_UAV_CONSUME_STRUCTURED` The shader resource is a consume-structured buffer.
`D3D11_SIT_UAV_RWSTRUCTURED_WITH_COUNTER` The shader resource is a read-and-write structured buffer that uses the built-in counter to append or consume.

Remarks

D3D_SHADER_INPUT_TYPE-typed values are specified in the **Type** member of the D3D11_SHADER_INPUT_BIND_DESC structure.

Requirements

Header	d3dcommon.h

See also

Common Version Enumerations

D3D_SHADER_VARIABLE_CLASS enumeration (d3dcommon.h)

Article • 01/31/20222 minutes to read

Values that identify the class of a shader variable.

> **Note**
>
> For programming with Direct3D 10, this API has a type alias that begins `D3D10_` instead of `D3D_`. These Direct3D 10 type aliases are defined in `d3d10.h`, `d3d10misc.h`, and `d3d10shader.h`.

Syntax

C++

```
typedef enum _D3D_SHADER_VARIABLE_CLASS {
  D3D_SVC_SCALAR = 0,
  D3D_SVC_VECTOR,
  D3D_SVC_MATRIX_ROWS,
  D3D_SVC_MATRIX_COLUMNS,
  D3D_SVC_OBJECT,
  D3D_SVC_STRUCT,
  D3D_SVC_INTERFACE_CLASS,
  D3D_SVC_INTERFACE_POINTER,
  D3D10_SVC_SCALAR,
  D3D10_SVC_VECTOR,
  D3D10_SVC_MATRIX_ROWS,
  D3D10_SVC_MATRIX_COLUMNS,
  D3D10_SVC_OBJECT,
  D3D10_SVC_STRUCT,
  D3D11_SVC_INTERFACE_CLASS,
  D3D11_SVC_INTERFACE_POINTER,
  D3D_SVC_FORCE_DWORD = 0x7fffffff
} D3D_SHADER_VARIABLE_CLASS;
```

Constants

`D3D_SVC_SCALAR` Value: *0* The shader variable is a scalar.
`D3D_SVC_VECTOR` The shader variable is a vector.
`D3D_SVC_MATRIX_ROWS` The shader variable is a row-major matrix.
`D3D_SVC_MATRIX_COLUMNS` The shader variable is a column-major matrix.
`D3D_SVC_OBJECT` The shader variable is an object.
`D3D_SVC_STRUCT` The shader variable is a structure.
`D3D_SVC_INTERFACE_CLASS` The shader variable is a class.
`D3D_SVC_INTERFACE_POINTER` The shader variable is an interface.
`D3D10_SVC_SCALAR` The shader variable is a scalar.
`D3D10_SVC_VECTOR` The shader variable is a vector.
`D3D10_SVC_MATRIX_ROWS` The shader variable is a row-major matrix.
`D3D10_SVC_MATRIX_COLUMNS` The shader variable is a column-major matrix.
`D3D10_SVC_OBJECT` The shader variable is an object.
`D3D10_SVC_STRUCT` The shader variable is a structure.
`D3D11_SVC_INTERFACE_CLASS` The shader variable is a class.
`D3D11_SVC_INTERFACE_POINTER` The shader variable is an interface.

`D3D_SVC_FORCE_DWORD` Value: *0x7fffffff* This value is not used by a programmer; it exists to force the enumeration to compile to 32 bits.

Remarks

The class of a shader variable is not a programming class; the class identifies the variable class such as scalar, vector, object, and so on. **D3D_SHADER_VARIABLE_CLASS**-typed values are specified in the **Class** member of the D3D11_SHADER_TYPE_DESC structure.

Requirements

Header	d3dcommon.h

See also

Common Version Enumerations

D3D_SHADER_VARIABLE_FLAGS enumeration (d3dcommon.h)

Article • 01/31/20222 minutes to read

Values that identify information about a shader variable.

> **Note**
>
> For programming with Direct3D 10, this API has a type alias that begins `D3D10_` instead of `D3D_`. These Direct3D 10 type aliases are defined in `d3d10.h`, `d3d10misc.h`, and `d3d10shader.h`.

Syntax

C++

```
typedef enum _D3D_SHADER_VARIABLE_FLAGS {
  D3D_SVF_USERPACKED = 1,
  D3D_SVF_USED = 2,
  D3D_SVF_INTERFACE_POINTER = 4,
  D3D_SVF_INTERFACE_PARAMETER = 8,
  D3D10_SVF_USERPACKED,
  D3D10_SVF_USED,
  D3D11_SVF_INTERFACE_POINTER,
  D3D11_SVF_INTERFACE_PARAMETER,
  D3D_SVF_FORCE_DWORD = 0x7fffffff
} D3D_SHADER_VARIABLE_FLAGS;
```

Constants

`D3D_SVF_USERPACKED` Value: *1* Indicates that the registers assigned to this shader variable were explicitly declared in shader code (instead of automatically assigned by the compiler).
`D3D_SVF_USED` Value: *2* Indicates that this variable is used by this shader. This value confirms that a particular shader variable (which can be common to many different shaders) is indeed used by a particular shader.

`D3D_SVF_INTERFACE_POINTER` Value: *4* Indicates that this variable is an interface.	
`D3D_SVF_INTERFACE_PARAMETER` Value: *8* Indicates that this variable is a parameter of an interface.	
`D3D10_SVF_USERPACKED` Indicates that the registers assigned to this shader variable were explicitly declared in shader code (instead of automatically assigned by the compiler).	
`D3D10_SVF_USED` Indicates that this variable is used by this shader. This value confirms that a particular shader variable (which can be common to many different shaders) is indeed used by a particular shader.	
`D3D11_SVF_INTERFACE_POINTER` Indicates that this variable is an interface.	
`D3D11_SVF_INTERFACE_PARAMETER` Indicates that this variable is a parameter of an interface.	
`D3D_SVF_FORCE_DWORD` Value: *0x7fffffff* This value is not used by a programmer; it exists to force the enumeration to compile to 32 bits.	

Remarks

A call to the ID3D11ShaderReflectionVariable::GetDesc method returns **D3D_SHADER_VARIABLE_FLAGS** values in the **uFlags** member of a D3D11_SHADER_VARIABLE_DESC structure.

Requirements

Header	d3dcommon.h

See also

Common Version Enumerations

D3D_SHADER_VARIABLE_TYPE enumeration (d3dcommon.h)

Article01/31/2022

Values that identify various data, texture, and buffer types that can be assigned to a shader variable.

> **Note**
>
> For programming with Direct3D 10, this API has a type alias that begins `D3D10_` instead of `D3D_`. These Direct3D 10 type aliases are defined in `d3d10.h`, `d3d10misc.h`, and `d3d10shader.h`.

Syntax

C++

```
typedef enum _D3D_SHADER_VARIABLE_TYPE {
  D3D_SVT_VOID = 0,
  D3D_SVT_BOOL = 1,
  D3D_SVT_INT = 2,
  D3D_SVT_FLOAT = 3,
  D3D_SVT_STRING = 4,
  D3D_SVT_TEXTURE = 5,
  D3D_SVT_TEXTURE1D = 6,
  D3D_SVT_TEXTURE2D = 7,
  D3D_SVT_TEXTURE3D = 8,
  D3D_SVT_TEXTURECUBE = 9,
  D3D_SVT_SAMPLER = 10,
  D3D_SVT_SAMPLER1D = 11,
  D3D_SVT_SAMPLER2D = 12,
  D3D_SVT_SAMPLER3D = 13,
  D3D_SVT_SAMPLERCUBE = 14,
  D3D_SVT_PIXELSHADER = 15,
  D3D_SVT_VERTEXSHADER = 16,
  D3D_SVT_PIXELFRAGMENT = 17,
  D3D_SVT_VERTEXFRAGMENT = 18,
  D3D_SVT_UINT = 19,
  D3D_SVT_UINT8 = 20,
  D3D_SVT_GEOMETRYSHADER = 21,
  D3D_SVT_RASTERIZER = 22,
  D3D_SVT_DEPTHSTENCIL = 23,
  D3D_SVT_BLEND = 24,
  D3D_SVT_BUFFER = 25,
  D3D_SVT_CBUFFER = 26,
```

```
    D3D_SVT_TBUFFER = 27,
    D3D_SVT_TEXTURE1DARRAY = 28,
    D3D_SVT_TEXTURE2DARRAY = 29,
    D3D_SVT_RENDERTARGETVIEW = 30,
    D3D_SVT_DEPTHSTENCILVIEW = 31,
    D3D_SVT_TEXTURE2DMS = 32,
    D3D_SVT_TEXTURE2DMSARRAY = 33,
    D3D_SVT_TEXTURECUBEARRAY = 34,
    D3D_SVT_HULLSHADER = 35,
    D3D_SVT_DOMAINSHADER = 36,
    D3D_SVT_INTERFACE_POINTER = 37,
    D3D_SVT_COMPUTESHADER = 38,
    D3D_SVT_DOUBLE = 39,
    D3D_SVT_RWTEXTURE1D = 40,
    D3D_SVT_RWTEXTURE1DARRAY = 41,
    D3D_SVT_RWTEXTURE2D = 42,
    D3D_SVT_RWTEXTURE2DARRAY = 43,
    D3D_SVT_RWTEXTURE3D = 44,
    D3D_SVT_RWBUFFER = 45,
    D3D_SVT_BYTEADDRESS_BUFFER = 46,
    D3D_SVT_RWBYTEADDRESS_BUFFER = 47,
    D3D_SVT_STRUCTURED_BUFFER = 48,
    D3D_SVT_RWSTRUCTURED_BUFFER = 49,
    D3D_SVT_APPEND_STRUCTURED_BUFFER = 50,
    D3D_SVT_CONSUME_STRUCTURED_BUFFER = 51,
    D3D_SVT_MIN8FLOAT = 52,
    D3D_SVT_MIN10FLOAT = 53,
    D3D_SVT_MIN16FLOAT = 54,
    D3D_SVT_MIN12INT = 55,
    D3D_SVT_MIN16INT = 56,
    D3D_SVT_MIN16UINT = 57,
    D3D_SVT_INT16,
    D3D_SVT_UINT16,
    D3D_SVT_FLOAT16,
    D3D_SVT_INT64,
    D3D_SVT_UINT64,
    D3D10_SVT_VOID,
    D3D10_SVT_BOOL,
    D3D10_SVT_INT,
    D3D10_SVT_FLOAT,
    D3D10_SVT_STRING,
    D3D10_SVT_TEXTURE,
    D3D10_SVT_TEXTURE1D,
    D3D10_SVT_TEXTURE2D,
    D3D10_SVT_TEXTURE3D,
    D3D10_SVT_TEXTURECUBE,
    D3D10_SVT_SAMPLER,
    D3D10_SVT_SAMPLER1D,
    D3D10_SVT_SAMPLER2D,
    D3D10_SVT_SAMPLER3D,
    D3D10_SVT_SAMPLERCUBE,
    D3D10_SVT_PIXELSHADER,
    D3D10_SVT_VERTEXSHADER,
    D3D10_SVT_PIXELFRAGMENT,
    D3D10_SVT_VERTEXFRAGMENT,
```

```
  D3D10_SVT_UINT,
  D3D10_SVT_UINT8,
  D3D10_SVT_GEOMETRYSHADER,
  D3D10_SVT_RASTERIZER,
  D3D10_SVT_DEPTHSTENCIL,
  D3D10_SVT_BLEND,
  D3D10_SVT_BUFFER,
  D3D10_SVT_CBUFFER,
  D3D10_SVT_TBUFFER,
  D3D10_SVT_TEXTURE1DARRAY,
  D3D10_SVT_TEXTURE2DARRAY,
  D3D10_SVT_RENDERTARGETVIEW,
  D3D10_SVT_DEPTHSTENCILVIEW,
  D3D10_SVT_TEXTURE2DMS,
  D3D10_SVT_TEXTURE2DMSARRAY,
  D3D10_SVT_TEXTURECUBEARRAY,
  D3D11_SVT_HULLSHADER,
  D3D11_SVT_DOMAINSHADER,
  D3D11_SVT_INTERFACE_POINTER,
  D3D11_SVT_COMPUTESHADER,
  D3D11_SVT_DOUBLE,
  D3D11_SVT_RWTEXTURE1D,
  D3D11_SVT_RWTEXTURE1DARRAY,
  D3D11_SVT_RWTEXTURE2D,
  D3D11_SVT_RWTEXTURE2DARRAY,
  D3D11_SVT_RWTEXTURE3D,
  D3D11_SVT_RWBUFFER,
  D3D11_SVT_BYTEADDRESS_BUFFER,
  D3D11_SVT_RWBYTEADDRESS_BUFFER,
  D3D11_SVT_STRUCTURED_BUFFER,
  D3D11_SVT_RWSTRUCTURED_BUFFER,
  D3D11_SVT_APPEND_STRUCTURED_BUFFER,
  D3D11_SVT_CONSUME_STRUCTURED_BUFFER,
  D3D_SVT_FORCE_DWORD = 0x7fffffff
} D3D_SHADER_VARIABLE_TYPE;
```

Constants

`D3D_SVT_VOID` Value: *0* The variable is a void pointer.
`D3D_SVT_BOOL` Value: *1* The variable is a boolean.
`D3D_SVT_INT` Value: *2* The variable is an integer.

`D3D_SVT_FLOAT` Value: *3* The variable is a floating-point number.
`D3D_SVT_STRING` Value: *4* The variable is a string.
`D3D_SVT_TEXTURE` Value: *5* The variable is a texture.
`D3D_SVT_TEXTURE1D` Value: *6* The variable is a 1D texture.
`D3D_SVT_TEXTURE2D` Value: *7* The variable is a 2D texture.
`D3D_SVT_TEXTURE3D` Value: *8* The variable is a 3D texture.
`D3D_SVT_TEXTURECUBE` Value: *9* The variable is a texture cube.
`D3D_SVT_SAMPLER` Value: *10* The variable is a sampler.
`D3D_SVT_SAMPLER1D` Value: *11* The variable is a 1D sampler.
`D3D_SVT_SAMPLER2D` Value: *12* The variable is a 2D sampler.
`D3D_SVT_SAMPLER3D` Value: *13* The variable is a 3D sampler.
`D3D_SVT_SAMPLERCUBE` Value: *14* The variable is a cube sampler.

`D3D_SVT_PIXELSHADER` Value: *15* The variable is a pixel shader.
`D3D_SVT_VERTEXSHADER` Value: *16* The variable is a vertex shader.
`D3D_SVT_PIXELFRAGMENT` Value: *17* The variable is a pixel fragment.
`D3D_SVT_VERTEXFRAGMENT` Value: *18* The variable is a vertex fragment.
`D3D_SVT_UINT` Value: *19* The variable is an unsigned integer.
`D3D_SVT_UINT8` Value: *20* The variable is an 8-bit unsigned integer.
`D3D_SVT_GEOMETRYSHADER` Value: *21* The variable is a geometry shader.
`D3D_SVT_RASTERIZER` Value: *22* The variable is a rasterizer-state object.
`D3D_SVT_DEPTHSTENCIL` Value: *23* The variable is a depth-stencil-state object.
`D3D_SVT_BLEND` Value: *24* The variable is a blend-state object.
`D3D_SVT_BUFFER` Value: *25* The variable is a buffer.
`D3D_SVT_CBUFFER` Value: *26* The variable is a constant buffer.

`D3D_SVT_TBUFFER` Value: *27* The variable is a texture buffer.
`D3D_SVT_TEXTURE1DARRAY` Value: *28* The variable is a 1D-texture array.
`D3D_SVT_TEXTURE2DARRAY` Value: *29* The variable is a 2D-texture array.
`D3D_SVT_RENDERTARGETVIEW` Value: *30* The variable is a render-target view.
`D3D_SVT_DEPTHSTENCILVIEW` Value: *31* The variable is a depth-stencil view.
`D3D_SVT_TEXTURE2DMS` Value: *32* The variable is a 2D-multisampled texture.
`D3D_SVT_TEXTURE2DMSARRAY` Value: *33* The variable is a 2D-multisampled-texture array.
`D3D_SVT_TEXTURECUBEARRAY` Value: *34* The variable is a texture-cube array.
`D3D_SVT_HULLSHADER` Value: *35* The variable holds a compiled hull-shader binary.
`D3D_SVT_DOMAINSHADER` Value: *36* The variable holds a compiled domain-shader binary.
`D3D_SVT_INTERFACE_POINTER` Value: *37* The variable is an interface.
`D3D_SVT_COMPUTESHADER` Value: *38* The variable holds a compiled compute-shader binary.

`D3D_SVT_DOUBLE` Value: *39* The variable is a double precision (64-bit) floating-point number.
`D3D_SVT_RWTEXTURE1D` Value: *40* The variable is a 1D read-and-write texture.
`D3D_SVT_RWTEXTURE1DARRAY` Value: *41* The variable is an array of 1D read-and-write textures.
`D3D_SVT_RWTEXTURE2D` Value: *42* The variable is a 2D read-and-write texture.
`D3D_SVT_RWTEXTURE2DARRAY` Value: *43* The variable is an array of 2D read-and-write textures.
`D3D_SVT_RWTEXTURE3D` Value: *44* The variable is a 3D read-and-write texture.
`D3D_SVT_RWBUFFER` Value: *45* The variable is a read-and-write buffer.
`D3D_SVT_BYTEADDRESS_BUFFER` Value: *46* The variable is a byte-address buffer.
`D3D_SVT_RWBYTEADDRESS_BUFFER` Value: *47* The variable is a read-and-write byte-address buffer.
`D3D_SVT_STRUCTURED_BUFFER` Value: *48* The variable is a structured buffer. For more information about structured buffer, see the **Remarks** section.
`D3D_SVT_RWSTRUCTURED_BUFFER` Value: *49* The variable is a read-and-write structured buffer.

`D3D_SVT_APPEND_STRUCTURED_BUFFER` Value: *50* The variable is an append structured buffer.
`D3D_SVT_CONSUME_STRUCTURED_BUFFER` Value: *51* The variable is a consume structured buffer.
`D3D_SVT_MIN8FLOAT` Value: *52* The variable is an 8-byte FLOAT.
`D3D_SVT_MIN10FLOAT` Value: *53* The variable is a 10-byte FLOAT.
`D3D_SVT_MIN16FLOAT` Value: *54* The variable is a 16-byte FLOAT.
`D3D_SVT_MIN12INT` Value: *55* The variable is a 12-byte INT.
`D3D_SVT_MIN16INT` Value: *56* The variable is a 16-byte INT.
`D3D_SVT_MIN16UINT` Value: *57* The variable is a 16-byte INT.
`D3D10_SVT_VOID` The variable is a void pointer.
`D3D10_SVT_BOOL` The variable is a boolean.
`D3D10_SVT_INT` The variable is an integer.
`D3D10_SVT_FLOAT` The variable is a floating-point number.
`D3D10_SVT_STRING` The variable is a string.
`D3D10_SVT_TEXTURE` The variable is a texture.

`D3D10_SVT_TEXTURE1D` The variable is a 1D texture.
`D3D10_SVT_TEXTURE2D` The variable is a 2D texture.
`D3D10_SVT_TEXTURE3D` The variable is a 3D texture.
`D3D10_SVT_TEXTURECUBE` The variable is a texture cube.
`D3D10_SVT_SAMPLER` The variable is a sampler.
`D3D10_SVT_SAMPLER1D` The variable is a 1D sampler.
`D3D10_SVT_SAMPLER2D` The variable is a 2D sampler.
`D3D10_SVT_SAMPLER3D` The variable is a 3D sampler.
`D3D10_SVT_SAMPLERCUBE` The variable is a cube sampler.
`D3D10_SVT_PIXELSHADER` The variable is a pixel shader.
`D3D10_SVT_VERTEXSHADER` The variable is a vertex shader.
`D3D10_SVT_PIXELFRAGMENT` The variable is a pixel fragment.
`D3D10_SVT_VERTEXFRAGMENT` The variable is a vertex fragment.
`D3D10_SVT_UINT` The variable is an unsigned integer.
`D3D10_SVT_UINT8` The variable is an 8-bit unsigned integer.
`D3D10_SVT_GEOMETRYSHADER` The variable is a geometry shader.

`D3D10_SVT_RASTERIZER` The variable is a rasterizer-state object.
`D3D10_SVT_DEPTHSTENCIL` The variable is a depth-stencil-state object.
`D3D10_SVT_BLEND` The variable is a blend-state object.
`D3D10_SVT_BUFFER` The variable is a buffer.
`D3D10_SVT_CBUFFER` The variable is a constant buffer.
`D3D10_SVT_TBUFFER` The variable is a texture buffer.
`D3D10_SVT_TEXTURE1DARRAY` The variable is a 1D-texture array.
`D3D10_SVT_TEXTURE2DARRAY` The variable is a 2D-texture array.
`D3D10_SVT_RENDERTARGETVIEW` The variable is a render-target view.
`D3D10_SVT_DEPTHSTENCILVIEW` The variable is a depth-stencil view.
`D3D10_SVT_TEXTURE2DMS` The variable is a 2D-multisampled texture.
`D3D10_SVT_TEXTURE2DMSARRAY` The variable is a 2D-multisampled-texture array.
`D3D10_SVT_TEXTURECUBEARRAY` The variable is a texture-cube array.
`D3D11_SVT_HULLSHADER` The variable holds a compiled hull-shader binary.
`D3D11_SVT_DOMAINSHADER` The variable holds a compiled domain-shader binary.
`D3D11_SVT_INTERFACE_POINTER` The variable is an interface.

`D3D11_SVT_COMPUTESHADER` The variable holds a compiled compute-shader binary.
`D3D11_SVT_DOUBLE` The variable is a double precision (64-bit) floating-point number.
`D3D11_SVT_RWTEXTURE1D` The variable is a 1D read-and-write texture.
`D3D11_SVT_RWTEXTURE1DARRAY` The variable is an array of 1D read-and-write textures.
`D3D11_SVT_RWTEXTURE2D` The variable is a 2D read-and-write texture.
`D3D11_SVT_RWTEXTURE2DARRAY` The variable is an array of 2D read-and-write textures.
`D3D11_SVT_RWTEXTURE3D` The variable is a 3D read-and-write texture.
`D3D11_SVT_RWBUFFER` The variable is a read-and-write buffer.
`D3D11_SVT_BYTEADDRESS_BUFFER` The variable is a byte-address buffer.
`D3D11_SVT_RWBYTEADDRESS_BUFFER` The variable is a read and write byte-address buffer.
`D3D11_SVT_STRUCTURED_BUFFER` The variable is a structured buffer.
`D3D11_SVT_RWSTRUCTURED_BUFFER` The variable is a read-and-write structured buffer.
`D3D11_SVT_APPEND_STRUCTURED_BUFFER` The variable is an append structured buffer.
`D3D11_SVT_CONSUME_STRUCTURED_BUFFER` The variable is a consume structured buffer.
`D3D_SVT_FORCE_DWORD` Value: *0x7fffffff* This value is not used by a programmer; it exists to force the enumeration to compile to 32 bits.

Remarks

A call to the ID3D11ShaderReflectionType::GetDesc method returns a **D3D_SHADER_VARIABLE_TYPE** value in the **Type** member of a D3D11_SHADER_TYPE_DESC structure.

The types in a structured buffer describe the structure of the elements in the buffer. The layout of these types generally match their C++ struct counterparts. The following examples show structured buffers:

```
struct mystruct {float4 val; uint ind;}; RWStructuredBuffer<mystruct> rwbuf;
                                         RWStructuredBuffer<float3> rwbuf2;
```

Requirements

Header	d3dcommon.h

See also

Common Version Enumerations

Shader Constants (Direct3D 10 Graphics)

Article • 01/06/2021 • 2 minutes to read

This section contains information about the following shader constants:

Constants	Description
D3D10_SHADER Constants	These constants identify HLSL compile options.

Related topics

Shader Reference

D3D10_SHADER Constants

Article • 01/06/2021 • 2 minutes to read

HLSL compile options.

#define	Description
D3D10_SHADER_AVOID_FLOW_CONTROL	Tell compiler to not allow flow-control (when possible).
D3D10_SHADER_DEBUG	Insert debug file/line/type/symbol information.
D3D10_SHADER_ENABLE_STRICTNESS	By default, the HLSL compiler disables strictness on deprecated syntax. Specifying this flag enables strictness which may not allow for legacy syntax.
D3D10_SHADER_ENABLE_BACKWARDS_COMPATIBILITY	This enables older shaders to compile to 4_0 targets.
D3D10_SHADER_FORCE_VS_SOFTWARE_NO_OPT	Compile a vertex shader for the next highest shader profile. This option turns debugging on (and optimizations off).
D3D10_SHADER_FORCE_PS_SOFTWARE_NO_OPT	Compile a pixel shader for the next highest shader profile. This option turns debugging on (and optimizations off).
D3D10_SHADER_IEEE_STRICTNESS	Enables IEEE strictness.
D3D10_SHADER_NO_PRESHADER	Disables Preshaders. Using this flag will cause the compiler to not pull out static expression for evaluation.
D3D10_SHADER_OPTIMIZATION_LEVEL0	Lowest optimization level. May produce slower code but will do so more quickly. This may be useful in a highly iterative shader development cycle.
D3D10_SHADER_OPTIMIZATION_LEVEL1	Second lowest optimization level.
D3D10_SHADER_OPTIMIZATION_LEVEL2	Second highest optimization level.

#define	Description
D3D10_SHADER_OPTIMIZATION_LEVEL3	Highest optimization level. Will produce best possible code but may take significantly longer to do so. This will be useful for final builds of an application where performance is the most important factor.
D3D10_SHADER_PACK_MATRIX_ROW_MAJOR	Unless explicitly specified, matrices will be packed in row-major order on input and output from the shader.
D3D10_SHADER_PACK_MATRIX_COLUMN_MAJOR	Unless explicitly specified, matrices will be packed in column-major order on input and output from the shader. This is generally more efficient, since it allows vector-matrix multiplication to be performed using a series of dot-products.
D3D10_SHADER_PARTIAL_PRECISION	Force all computations to be done with partial precision; this may run faster on some hardware.
D3D10_SHADER_PREFER_FLOW_CONTROL	Tell compiler to use flow-control (when possible).
D3D10_SHADER_SKIP_OPTIMIZATION	Skip optimization during code generation; generally recommended for debug only.
D3D10_SHADER_SKIP_VALIDATION	Do not validate the generated code against known capabilities and constraints. Only use this with shaders that have been successfully compiled in the past. Shaders are always validated by DirectX before they are set to the device.
D3D10_SHADER_WARNINGS_ARE_ERRORS	Inform the HLSL compiler to treat all warnings as errors when compiling the shader code. For new shader code, you should use this option so you can resolve all warnings and ensure the fewest possible hard-to-find code defects.

These constants are defined as macros in d3d10shader.h.

Related topics

Shader Constants

Effect Reference (Direct3D 10 Graphics)

Article • 01/06/2021 • 2 minutes to read

The Direct3D API defines several API elements to help you create and manage the effect system.

- Interfaces
- Functions
- Structures
- Constants

The effect system encapsulates state assignments in an effect. Each effect is a collection of pipeline state which can be broken down into state assignments using state blocks, resources, and shaders. Depending on the size of an effect, you can choose to implement one in an ASCII text string or an effect file (.fx). To organize information in an effect, see the effect format.

Having designed an effect, use the effect API to set state in the device during rendering. As well, the effect system supports a series of reflection API's for getting and setting effect data. For more information, see Effect System Interfaces (Direct3D 10).

Related topics

Direct3D Reference

Effect Interfaces (Direct3D 10)

Article • 01/06/2021 • 2 minutes to read

This section contains information about the following effect-system interfaces:

Interfaces	Description
ID3D10EffectBlendVariable Interface	Accesses blend state.
ID3D10EffectConstantBuffer Interface	Accesses a texture-buffer or a constant-buffer.
ID3D10EffectDepthStencilVariable Interface	Accesses depth-stencil state.
ID3D10EffectDepthStencilViewVariable Interface	Accesses a depth-stencil view.
ID3D10Effect Interface	Encapsulates pipeline state in one or more rendering techniques.
ID3D10Include Interface	User-implemented methods for reading include files.
ID3D10EffectMatrixVariable Interface	Accesses a matrix.
ID3D10EffectPass Interface	Encapsulates effect state in a pass.
ID3D10EffectPool Interface	Identifies shared-effect variables.
ID3D10EffectRasterizerVariable Interface	Accesses rasterizer state.
ID3D10EffectRenderTargetViewVariable Interface	Accesses a render target.
ID3D10EffectSamplerVariable Interface	Accesses sampler state.
ID3D10EffectScalarVariable Interface	Accesses a scalar variable.
ID3D10EffectShaderResourceVariable Interface	Accesses a shader resource.
ID3D10EffectShaderVariable Interface	Accesses a shader variable.
ID3D10EffectStringVariable Interface	Accesses a string.
ID3D10EffectTechnique Interface	Encapsulates one or more passes.
ID3D10EffectType Interface	Implements methods for accessing effect variables.
ID3D10EffectVectorVariable Interface	Accesses a vector.

There are two kinds of interfaces in the effect framework: rendering interfaces for rendering an effect and reflection interfaces for getting and setting effect variables with the API. All reflection interfaces derive from **ID3D10EffectVariable Interface**.

Related topics

Effect Reference

ID3D10Effect interface (d3d10effect.h)

Article • 08/19/20222 minutes to read

An **ID3D10Effect** interface manages a set of state objects, resources, and shaders for implementing a rendering effect.

Inheritance

The **ID3D10Effect** interface inherits from the IUnknown interface. **ID3D10Effect** also has these types of members:

Methods

The **ID3D10Effect** interface has these methods.

ID3D10Effect::GetConstantBufferByIndex The ID3D10Effect::GetConstantBufferByIndex (d3d10effect.h) method gets a constant buffer by index.
ID3D10Effect::GetConstantBufferByName Get a constant buffer by name. (ID3D10Effect.GetConstantBufferByName)
ID3D10Effect::GetDesc Get an effect description.
ID3D10Effect::GetDevice Get the device that created the effect.
ID3D10Effect::GetTechniqueByIndex Get a technique by index.
ID3D10Effect::GetTechniqueByName Get a technique by name.
ID3D10Effect::GetVariableByIndex Get a variable by index.

ID3D10Effect::GetVariableByName	Get a variable by name.
ID3D10Effect::GetVariableBySemantic	Get a variable by semantic.
ID3D10Effect::IsOptimized	Test an effect to see if the reflection metadata has been removed from memory.
ID3D10Effect::IsPool	Test an effect to see if it is part of a memory pool.
ID3D10Effect::IsValid	Test an effect to see if it contains valid syntax.
ID3D10Effect::Optimize	Minimize the amount of memory required for an effect.

Remarks

An effect is created by calling D3D10CreateEffectFromMemory.

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. For more information, see Effects (Direct3D 10).

Note

If you call **QueryInterface** on an **ID3D10Effect** 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

Target Platform	Windows
Header	d3d10effect.h

See also

Effect Interfaces (Direct3D 10)

ID3D10Effect::GetConstantBufferByIndex method (d3d10effect.h)

Article08/19/2022

Get a constant buffer by index.

Syntax

```
C++

ID3D10EffectConstantBuffer * GetConstantBufferByIndex(
  [in] UINT Index
);
```

Parameters

`[in] Index`

Type: **UINT**

A zero-based index.

Return value

Type: **ID3D10EffectConstantBuffer***

A pointer to a ID3D10EffectConstantBuffer Interface.

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.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10Effect Interface

ID3D10Effect::GetConstantBufferByName method (d3d10effect.h)

Article • 07/27/20222 minutes to read

Get a constant buffer by name.

Syntax

```
C++

ID3D10EffectConstantBuffer * GetConstantBufferByName(
  [in] LPCSTR Name
);
```

Parameters

`[in] Name`

Type: **LPCSTR**

The constant-buffer name.

Return value

Type: **ID3D10EffectConstantBuffer***

A pointer to the constant buffer indicated by the Name. See ID3D10EffectConstantBuffer.

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.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10Effect Interface

ID3D10Effect::GetDesc method (d3d10effect.h)

Article10/13/2021

Get an effect description.

Syntax

```
C++

HRESULT GetDesc(
  [in, out] D3D10_EFFECT_DESC *pDesc
);
```

Parameters

`[in, out] pDesc`

Type: **D3D10_EFFECT_DESC***

A pointer to an effect description (see D3D10_EFFECT_DESC).

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 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.

Requirements

Target Platform	Windows

Header	d3d10effect.h

See also

ID3D10Effect Interface

ID3D10Effect::GetDevice method (d3d10effect.h)

Article10/13/2021

Get the device that created the effect.

Syntax

```
C++

HRESULT GetDevice(
  [out] ID3D10Device **ppDevice
);
```

Parameters

`[out] ppDevice`

Type: **ID3D10Device****

A pointer to an ID3D10Device Interface.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Remarks

An effect is created for a specific device, by calling a function such as D3DX10CreateEffectFromFile.

Requirements

Target Platform	Windows

Header	d3d10effect.h

See also

ID3D10Effect Interface

ID3D10Effect::GetTechniqueByIndex method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get a technique by index.

Syntax

```
C++

ID3D10EffectTechnique * GetTechniqueByIndex(
  [in] UINT Index
);
```

Parameters

`[in] Index`

Type: **UINT**

A zero-based index.

Return value

Type: **ID3D10EffectTechnique***

A pointer to an ID3D10EffectTechnique Interface.

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. For more about techniques, see techniques and passes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10Effect Interface

ID3D10Effect::GetTechniqueByName method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get a technique by name.

Syntax

```
C++

ID3D10EffectTechnique * GetTechniqueByName(
  [in] LPCSTR Name
);
```

Parameters

`[in] Name`

Type: **LPCSTR**

The name of the technique.

Return value

Type: **ID3D10EffectTechnique***

A pointer to an ID3D10EffectTechnique Interface, or **NULL** if the technique is not found.

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. For more about techniques, see techniques and passes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10Effect Interface

ID3D10Effect::GetVariableByIndex method (d3d10effect.h)

Article10/13/2021

Get a variable by index.

Syntax

```
C++

ID3D10EffectVariable * GetVariableByIndex(
  [in] UINT Index
);
```

Parameters

`[in] Index`

Type: **UINT**

A zero-based index.

Return value

Type: **ID3D10EffectVariable***

A pointer to a ID3D10EffectVariable Interface.

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 ID3D10Effect::IsValid to verify whether or not the index exists.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10Effect Interface

ID3D10Effect::GetVariableByName method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get a variable by name.

Syntax

```
C++

ID3D10EffectVariable * GetVariableByName(
  [in] LPCSTR Name
);
```

Parameters

`[in] Name`

Type: **LPCSTR**

The variable name.

Return value

Type: **ID3D10EffectVariable***

A pointer to an ID3D10EffectVariable Interface.

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 if a variable is not found; you can call ID3D10Effect::IsValid to verify whether or not the name exists.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10Effect Interface

ID3D10Effect::GetVariableBySemantic method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get a variable by semantic.

Syntax

```
C++

ID3D10EffectVariable * GetVariableBySemantic(
  [in] LPCSTR Semantic
);
```

Parameters

`[in] Semantic`

Type: **LPCSTR**

The semantic name.

Return value

Type: **ID3D10EffectVariable***

A pointer to the effect variable indicated by the Semantic. See ID3D10EffectVariable Interface.

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 ID3D10Effect::IsValid to verify whether or not the semantic exists.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10Effect Interface

ID3D10Effect::IsOptimized method (d3d10effect.h)

Article • 06/29/20212 minutes to read

Test an effect to see if the reflection metadata has been removed from memory.

Syntax

```
C++

BOOL IsOptimized();
```

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 ID3D10Effect::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.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10Effect Interface

ID3D10Effect::IsPool method (d3d10effect.h)

Article • 06/29/20212 minutes to read

Test an effect to see if it is part of a memory pool.

Syntax

```
C++

BOOL IsPool();
```

Return value

Type: **BOOL**

TRUE if the effect is pooled; otherwise **FALSE**. See ID3D10EffectPool Interface.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10Effect Interface

ID3D10Effect::IsValid method (d3d10effect.h)

Article • 06/29/20212 minutes to read

Test an effect to see if it contains valid syntax.

Syntax

```
C++

BOOL IsValid();
```

Return value

Type: **BOOL**

TRUE if the code syntax is valid; otherwise **FALSE**.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10Effect Interface

ID3D10Effect::Optimize method (d3d10effect.h)

Article • 06/29/20212 minutes to read

Minimize the amount of memory required for an effect.

Syntax

```
C++

HRESULT Optimize();
```

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 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 **ID3D10Effect::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.

- ID3D10Effect::GetConstantBufferByIndex
- ID3D10Effect::GetConstantBufferByName
- ID3D10Effect::GetDesc
- ID3D10Effect::GetDevice
- ID3D10Effect::GetTechniqueByIndex
- ID3D10Effect::GetTechniqueByName
- ID3D10Effect::GetVariableByIndex
- ID3D10Effect::GetVariableByName
- ID3D10Effect::GetVariableBySemantic

Note that references retrieved with these methods before calling **ID3D10Effect::Optimize** are still valid after **ID3D10Effect::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.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10Effect Interface

ID3D10EffectBlendVariable interface (d3d10effect.h)

Article • 07/22/20212 minutes to read

The blend-variable interface accesses blend state.

Inheritance

The **ID3D10EffectBlendVariable** interface inherits from ID3D10EffectVariable. **ID3D10EffectBlendVariable** also has these types of members:

Methods

The **ID3D10EffectBlendVariable** interface has these methods.

ID3D10EffectBlendVariable::GetBackingStore Get a pointer to a blend-state variable.
ID3D10EffectBlendVariable::GetBlendState Get a pointer to a blend-state interface.

Remarks

An **ID3D10EffectBlendVariable 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. For examples, see Two Ways to Get the State in an Effect Variable.

Requirements

Target Platform	Windows

Header	d3d10effect.h

See also

Effect Interfaces (Direct3D 10)

ID3D10EffectVariable

ID3D10EffectBlendVariable::GetBackingStore method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get a pointer to a blend-state variable.

Syntax

C++

```
HRESULT GetBackingStore(
  [in] UINT             Index,
  [in] D3D10_BLEND_DESC *pBlendDesc
);
```

Parameters

`[in] Index`

Type: **UINT**

Index into an array of blend-state descriptions. If there is only one blend-state variable in the effect, use 0.

`[in] pBlendDesc`

Type: **D3D10_BLEND_DESC***

A pointer to a blend-state description (see D3D10_BLEND_DESC).

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 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 used to

recreate the variable when necessary.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectBlendVariable Interface

ID3D10EffectBlendVariable::GetBlendState method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get a pointer to a blend-state interface.

Syntax

```
C++

HRESULT GetBlendState(
  [in]  UINT             Index,
  [out] ID3D10BlendState **ppBlendState
);
```

Parameters

`[in] Index`

Type: **UINT**

Index into an array of blend-state interfaces. If there is only one blend-state interface, use 0.

`[out] ppBlendState`

Type: **ID3D10BlendState****

The address of a pointer to a blend-state interface (see ID3D10BlendState Interface).

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectBlendVariable Interface

ID3D10EffectConstantBuffer interface (d3d10effect.h)

Article • 07/22/20212 minutes to read

A constant-buffer interface accesses constant buffers or texture buffers.

Inheritance

The **ID3D10EffectConstantBuffer** interface inherits from ID3D10EffectVariable. **ID3D10EffectConstantBuffer** also has these types of members:

Methods

The **ID3D10EffectConstantBuffer** interface has these methods.

ID3D10EffectConstantBuffer::GetConstantBuffer Get a constant-buffer.
ID3D10EffectConstantBuffer::GetTextureBuffer Get a texture-buffer.
ID3D10EffectConstantBuffer::SetConstantBuffer Set a constant-buffer.
ID3D10EffectConstantBuffer::SetTextureBuffer Set a 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.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

Effect Interfaces (Direct3D 10)

ID3D10EffectVariable

ID3D10EffectConstantBuffer::GetConstantBuffer method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get a constant-buffer.

Syntax

```
C++

HRESULT GetConstantBuffer(
  [out] ID3D10Buffer **ppConstantBuffer
);
```

Parameters

`[out] ppConstantBuffer`

Type: **ID3D10Buffer****

The address of a pointer to a constant-buffer interface. See ID3D10Buffer Interface.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectConstantBuffer Interface

ID3D10EffectConstantBuffer::GetTextureBuffer method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get a texture-buffer.

Syntax

```
C++

HRESULT GetTextureBuffer(
  [out] ID3D10ShaderResourceView **ppTextureBuffer
);
```

Parameters

`[out] ppTextureBuffer`

Type: **ID3D10ShaderResourceView****

The address of a pointer to a shader-resource-view interface for accessing a texture buffer. See ID3D10ShaderResourceView Interface.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectConstantBuffer Interface

ID3D10EffectConstantBuffer::SetConstantBuffer method (d3d10effect.h)

Article10/13/2021

Set a constant-buffer.

Syntax

```
C++

HRESULT SetConstantBuffer(
  [in] ID3D10Buffer *pConstantBuffer
);
```

Parameters

`[in] pConstantBuffer`

Type: **ID3D10Buffer***

A pointer to a constant-buffer interface. See ID3D10Buffer Interface.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectConstantBuffer Interface

ID3D10EffectConstantBuffer::SetTexture Buffer method (d3d10effect.h)

Article10/13/2021

Set a texture-buffer.

Syntax

```
C++

HRESULT SetTextureBuffer(
  [in] ID3D10ShaderResourceView *pTextureBuffer
);
```

Parameters

`[in] pTextureBuffer`

Type: **ID3D10ShaderResourceView***

A pointer to a shader-resource-view interface for accessing a texture buffer.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectConstantBuffer Interface

ID3D10EffectDepthStencilVariable interface (d3d10effect.h)

Article • 07/22/20212 minutes to read

A depth-stencil-variable interface accesses depth-stencil state.

Inheritance

The **ID3D10EffectDepthStencilVariable** interface inherits from ID3D10EffectVariable. **ID3D10EffectDepthStencilVariable** also has these types of members:

Methods

The **ID3D10EffectDepthStencilVariable** interface has these methods.

ID3D10EffectDepthStencilVariable::GetBackingStore Get a pointer to a variable that contains depth-stencil state.
ID3D10EffectDepthStencilVariable::GetDepthStencilState Get a pointer to a depth-stencil interface.

Remarks

An **ID3D10EffectDepthStencilVariable 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. For examples, see Two Ways to Get the State in an Effect Variable.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

Effect Interfaces (Direct3D 10)

ID3D10EffectVariable

ID3D10EffectDepthStencilVariable::GetBackingStore method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get a pointer to a variable that contains depth-stencil state.

Syntax

```
C++

HRESULT GetBackingStore(
  [in] UINT                      Index,
  [in] D3D10_DEPTH_STENCIL_DESC *pDepthStencilDesc
);
```

Parameters

`[in] 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.

`[in] pDepthStencilDesc`

Type: **D3D10_DEPTH_STENCIL_DESC***

A pointer to a depth-stencil-state description (see D3D10_DEPTH_STENCIL_DESC).

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 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 used to

recreate the variable when necessary.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectDepthStencilVariable Interface

ID3D10EffectDepthStencilVariable::GetDepthStencilState method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get a pointer to a depth-stencil interface.

Syntax

C++

```
HRESULT GetDepthStencilState(
  [in]  UINT                    Index,
  [out] ID3D10DepthStencilState **ppDepthStencilState
);
```

Parameters

`[in] Index`

Type: **UINT**

Index into an array of depth-stencil interfaces. If there is only one depth-stencil interface, use 0.

`[out] ppDepthStencilState`

Type: **ID3D10DepthStencilState****

The address of a pointer to a blend-state interface (see ID3D10DepthStencilState Interface).

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectDepthStencilVariable Interface

ID3D10EffectDepthStencilViewVariable interface (d3d10effect.h)

Article07/22/2021

A depth-stencil-view-variable interface accesses a depth-stencil view.

Inheritance

The **ID3D10EffectDepthStencilViewVariable** interface inherits from ID3D10EffectVariable. **ID3D10EffectDepthStencilViewVariable** also has these types of members:

Methods

The **ID3D10EffectDepthStencilViewVariable** interface has these methods.

ID3D10EffectDepthStencilViewVariable::GetDepthStencil Get a depth-stencil-view resource.
ID3D10EffectDepthStencilViewVariable::GetDepthStencilArray Get an array of depth-stencil-view resources.
ID3D10EffectDepthStencilViewVariable::SetDepthStencil Set a depth-stencil-view resource.
ID3D10EffectDepthStencilViewVariable::SetDepthStencilArray Set an array of depth-stencil-view resources.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

Effect Interfaces (Direct3D 10)

ID3D10EffectVariable

ID3D10EffectDepthStencilViewVariable::GetDepthStencil method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get a depth-stencil-view resource.

Syntax

C++

```
HRESULT GetDepthStencil(
  [out] ID3D10DepthStencilView **ppResource
);
```

Parameters

`[out] ppResource`

Type: **ID3D10DepthStencilView****

The address of a pointer to a depth-stencil-view interface. See ID3D10DepthStencilView Interface.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h (include D3d10)

See also

ID3D10EffectDepthStencilViewVariable Interface

ID3D10EffectDepthStencilViewVariable::GetDepthStencilArray method (d3d10effect.h)

Article10/13/2021

Get an array of depth-stencil-view resources.

Syntax

```
C++

HRESULT GetDepthStencilArray(
  [out] ID3D10DepthStencilView **ppResources,
  [in]  UINT                   Offset,
  [in]  UINT                   Count
);
```

Parameters

`[out] ppResources`

Type: **ID3D10DepthStencilView****

A pointer to an array of depth-stencil-view interfaces. See ID3D10DepthStencilView Interface.

`[in] Offset`

Type: **UINT**

The zero-based array index to get the first interface.

`[in] Count`

Type: **UINT**

The number of elements in the array.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h (include D3d10)

See also

ID3D10EffectDepthStencilViewVariable Interface

ID3D10EffectDepthStencilViewVariable::SetDepthStencil method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Set a depth-stencil-view resource.

Syntax

C++

```
HRESULT SetDepthStencil(
  [in] ID3D10DepthStencilView *pResource
);
```

Parameters

`[in] pResource`

Type: **ID3D10DepthStencilView***

A pointer to a depth-stencil-view interface. See ID3D10DepthStencilView Interface.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h (include D3d10)

See also

ID3D10EffectDepthStencilViewVariable Interface

ID3D10EffectDepthStencilViewVariable::SetDepthStencilArray method (d3d10effect.h)

Article10/13/2021

Set an array of depth-stencil-view resources.

Syntax

```
C++

HRESULT SetDepthStencilArray(
  [out] ID3D10DepthStencilView **ppResources,
  [in]  UINT                   Offset,
  [in]  UINT                   Count
);
```

Parameters

`[out] ppResources`

Type: **ID3D10DepthStencilView****

A pointer to an array of depth-stencil-view interfaces. See ID3D10DepthStencilView Interface.

`[in] Offset`

Type: **UINT**

The zero-based array index to set the first interface.

`[in] Count`

Type: **UINT**

The number of elements in the array.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h (include D3d10)

See also

ID3D10EffectDepthStencilViewVariable Interface

ID3D10EffectMatrixVariable interface (d3d10effect.h)

Article • 07/22/20212 minutes to read

A matrix-variable interface accesses a matrix.

Inheritance

The **ID3D10EffectMatrixVariable** interface inherits from ID3D10EffectVariable. **ID3D10EffectMatrixVariable** also has these types of members:

Methods

The **ID3D10EffectMatrixVariable** interface has these methods.

ID3D10EffectMatrixVariable::GetMatrix Get a matrix.
ID3D10EffectMatrixVariable::GetMatrixArray Get an array of matrices.
ID3D10EffectMatrixVariable::GetMatrixTranspose Transpose and get a floating-point matrix.
ID3D10EffectMatrixVariable::GetMatrixTransposeArray Transpose and get an array of floating-point matrices.
ID3D10EffectMatrixVariable::SetMatrix Set a floating-point matrix.
ID3D10EffectMatrixVariable::SetMatrixArray Set an array of floating-point matrices.
ID3D10EffectMatrixVariable::SetMatrixTranspose Transpose and set a floating-point matrix.

ID3D10EffectMatrixVariable::SetMatrixTransposeArray Transpose and set an array of floating-point matrices.	

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

Effect Interfaces (Direct3D 10)

ID3D10EffectVariable

ID3D10EffectMatrixVariable::GetMatrix method (d3d10effect.h)

Article10/13/2021

Get a matrix.

Syntax

```
C++

HRESULT GetMatrix(
  [out] float *pData
);
```

Parameters

`[out] pData`

Type: **float***

A pointer to the first element in a matrix.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectMatrixVariable Interface

ID3D10EffectMatrixVariable::GetMatrixArray method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get an array of matrices.

Syntax

```
C++

HRESULT GetMatrixArray(
  [out] float *pData,
  [in]  UINT  Offset,
  [in]  UINT  Count
);
```

Parameters

`[out] pData`

Type: **float***

A pointer to the first element of the first matrix in an array of matrices.

`[in] Offset`

Type: **UINT**

The offset (in number of matrices) between the start of the array and the first matrix returned.

`[in] Count`

Type: **UINT**

The number of matrices in the returned array.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectMatrixVariable Interface

ID3D10EffectMatrixVariable::GetMatrixTranspose method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Transpose and get a floating-point matrix.

Syntax

```
C++

HRESULT GetMatrixTranspose(
  [out] float *pData
);
```

Parameters

`[out] pData`

Type: **float***

A pointer to the first element of a transposed matrix.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Remarks

Transposing a matrix will rearrange the data order from row-column order to column-row order (or vice versa).

Requirements

Target Platform	Windows

Header	d3d10effect.h

See also

ID3D10EffectMatrixVariable Interface

ID3D10EffectMatrixVariable::GetMatrixTransposeArray method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Transpose and get an array of floating-point matrices.

Syntax

C++

```
HRESULT GetMatrixTransposeArray(
  [out] float *pData,
  [in]  UINT  Offset,
  [in]  UINT  Count
);
```

Parameters

`[out] pData`

Type: **float***

A pointer to the first element of an array of transposed matrices.

`[in] Offset`

Type: **UINT**

The offset (in number of matrices) between the start of the array and the first matrix to get.

`[in] Count`

Type: **UINT**

The number of matrices in the array to get.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Remarks

Transposing a matrix will rearrange the data order from row-column order to column-row order (or vice versa).

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectMatrixVariable Interface

ID3D10EffectMatrixVariable::SetMatrix method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Set a floating-point matrix.

Syntax

```
C++

HRESULT SetMatrix(
  [in] float *pData
);
```

Parameters

`[in] pData`

Type: **float***

A pointer to the first element in the matrix.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectMatrixVariable Interface

ID3D10EffectMatrixVariable::SetMatrixArray method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Set an array of floating-point matrices.

Syntax

```
C++

HRESULT SetMatrixArray(
  [in] float *pData,
  [in] UINT  Offset,
  [in] UINT  Count
);
```

Parameters

`[in] pData`

Type: **float***

A pointer to the first matrix.

`[in] Offset`

Type: **UINT**

The number of matrix elements to skip from the start of the array.

`[in] Count`

Type: **UINT**

The number of elements to set.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectMatrixVariable Interface

ID3D10EffectMatrixVariable::SetMatrixTranspose method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Transpose and set a floating-point matrix.

Syntax

```
C++

HRESULT SetMatrixTranspose(
  [in] float *pData
);
```

Parameters

`[in] pData`

Type: **float***

A pointer to the first element of a matrix.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Remarks

Transposing a matrix will rearrange the data order from row-column order to column-row order (or vice versa).

Requirements

Target Platform	Windows

Header	d3d10effect.h

See also

ID3D10EffectMatrixVariable Interface

ID3D10EffectMatrixVariable::SetMatrixTransposeArray method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Transpose and set an array of floating-point matrices.

Syntax

C++

```
HRESULT SetMatrixTransposeArray(
  [in] float *pData,
  [in] UINT  Offset,
  [in] UINT  Count
);
```

Parameters

`[in] pData`

Type: **float***

A pointer to an array of matrices.

`[in] Offset`

Type: **UINT**

The offset (in number of matrices) between the start of the array and the first matrix to set.

`[in] Count`

Type: **UINT**

The number of matrices in the array to set.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Remarks

Transposing a matrix will rearrange the data order from row-column order to column-row order (or vice versa).

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectMatrixVariable Interface

ID3D10EffectPass interface (d3d10effect.h)

Article • 08/19/20222 minutes to read

A pass interface encapsulates state assignments within a technique.

The lifetime of an **ID3D10EffectPass** object is equal to the lifetime of its parent ID3D10Effect object.

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.
GetDesc	Get a pass description.
GetGeometryShaderDesc	Get a geometry-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.

Methods

The **ID3D10EffectPass** interface has these methods.

ID3D10EffectPass::Apply Set the state contained in a pass to the device.
ID3D10EffectPass::ComputeStateBlockMask Generate a mask for allowing/preventing state changes.

ID3D10EffectPass::GetAnnotationByIndex The ID3D10EffectPass::GetAnnotationByIndex (d3d10effect.h) method gets an annotation by index.	
ID3D10EffectPass::GetAnnotationByName Get an annotation by name. (ID3D10EffectPass.GetAnnotationByName)	
ID3D10EffectPass::GetDesc Get a pass description.	
ID3D10EffectPass::GetGeometryShaderDesc Get a geometry-shader description.	
ID3D10EffectPass::GetPixelShaderDesc Get a pixel-shader description.	
ID3D10EffectPass::GetVertexShaderDesc Get a vertex-shader description.	
ID3D10EffectPass::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; the syntax for a technique is shown in Effect Technique Syntax (Direct3D 10).

To get an effect-pass interface, call a method like ID3D10EffectTechnique::GetPassByName.

Requirements

Target Platform	Windows

Header	d3d10effect.h

See also

Effect Interfaces (Direct3D 10)

ID3D10EffectPass::Apply method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Set the state contained in a pass to the device.

Syntax

```
C++

HRESULT Apply(
  [in] UINT Flags
);
```

Parameters

`[in] Flags`

Type: **UINT**

Unused.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10Effect Interface

ID3D10EffectPass

ID3D10EffectPass::ComputeStateBlockMask method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Generate a mask for allowing/preventing state changes.

Syntax

```
C++

HRESULT ComputeStateBlockMask(
  [in] D3D10_STATE_BLOCK_MASK *pStateBlockMask
);
```

Parameters

`[in] pStateBlockMask`

Type: **D3D10_STATE_BLOCK_MASK***

A pointer to a state-block mask (see D3D10_STATE_BLOCK_MASK).

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10Effect Interface

ID3D10EffectPass

ID3D10EffectPass::GetAnnotationByIndex method (d3d10effect.h)

Article • 08/19/20222 minutes to read

Get an annotation by index.

Syntax

```
C++

ID3D10EffectVariable * GetAnnotationByIndex(
  [in] UINT Index
);
```

Parameters

`[in] Index`

Type: **UINT**

A zero-based index.

Return value

Type: **ID3D10EffectVariable***

A pointer to an ID3D10EffectVariable Interface.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10Effect Interface

ID3D10EffectPass

ID3D10EffectPass::GetAnnotationByName method (d3d10effect.h)

Article07/27/2022

Get an annotation by name.

Syntax

```
C++

ID3D10EffectVariable * GetAnnotationByName(
  [in] LPCSTR Name
);
```

Parameters

`[in] Name`

Type: **LPCSTR**

The name of the annotation.

Return value

Type: **ID3D10EffectVariable***

A pointer to an ID3D10EffectVariable Interface.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10Effect Interface

ID3D10EffectPass

ID3D10EffectPass::GetDesc method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get a pass description.

Syntax

```
C++

HRESULT GetDesc(
  [in] D3D10_PASS_DESC *pDesc
);
```

Parameters

`[in] pDesc`

Type: **D3D10_PASS_DESC***

A pointer to a pass description (see D3D10_PASS_DESC).

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 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. See techniques and passes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10Effect Interface

ID3D10EffectPass

ID3D10EffectPass::GetGeometryShaderDesc method (d3d10effect.h)

Article10/13/2021

Get a geometry-shader description.

Syntax

```
C++

HRESULT GetGeometryShaderDesc(
  [in] D3D10_PASS_SHADER_DESC *pDesc
);
```

Parameters

`[in] pDesc`

Type: **D3D10_PASS_SHADER_DESC***

A pointer to a geometry-shader description (see D3D10_PASS_SHADER_DESC).

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Remarks

An effect pass can contain render state assignments and shader object assignments.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10Effect Interface

ID3D10EffectPass

ID3D10EffectPass::GetPixelShaderDesc method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get a pixel-shader description.

Syntax

```
C++

HRESULT GetPixelShaderDesc(
  [in] D3D10_PASS_SHADER_DESC *pDesc
);
```

Parameters

`[in] pDesc`

Type: **D3D10_PASS_SHADER_DESC***

A pointer to a pixel-shader description (see D3D10_PASS_SHADER_DESC).

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Remarks

An effect pass can contain render state assignments and shader object assignments.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10Effect Interface

ID3D10EffectPass

ID3D10EffectPass::GetVertexShaderDesc method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get a vertex-shader description.

Syntax

```
C++

HRESULT GetVertexShaderDesc(
  [in] D3D10_PASS_SHADER_DESC *pDesc
);
```

Parameters

`[in] pDesc`

Type: **D3D10_PASS_SHADER_DESC***

A pointer to a vertex-shader description (see D3D10_PASS_SHADER_DESC).

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Remarks

An effect pass can contain render state assignments and shader object assignments.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10Effect Interface

ID3D10EffectPass

ID3D10EffectPass::IsValid method (d3d10effect.h)

Article • 06/29/20212 minutes to read

Test a pass to see if it contains valid syntax.

Syntax

```
C++

BOOL IsValid();
```

Return value

Type: **BOOL**

TRUE if the code syntax is valid; otherwise **FALSE**.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10Effect Interface

ID3D10EffectPass

ID3D10EffectPool interface (d3d10effect.h)

Article • 07/22/20212 minutes to read

A pool interface represents a common memory space (or pool) for sharing variables between effects.

Inheritance

The **ID3D10EffectPool** interface inherits from the IUnknown interface. **ID3D10EffectPool** also has these types of members:

Methods

The **ID3D10EffectPool** interface has these methods.

ID3D10EffectPool::AsEffect Get the effect that created the effect pool.

Remarks

To create an effect pool, call a function like D3DX10CreateEffectPoolFromFile. Effect pools can improve performance by reducing the number of API calls required to make state changes (see Using Effect Pools).

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

Effect Interfaces (Direct3D 10)

ID3D10EffectPool::AsEffect method (d3d10effect.h)

Article • 06/29/20212 minutes to read

Get the effect that created the effect pool.

Syntax

```
C++

ID3D10Effect * AsEffect();
```

Return value

Type: **ID3D10Effect***

A pointer to an ID3D10Effect Interface interface.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectPool Interface

ID3D10EffectRasterizerVariable interface (d3d10effect.h)

Article • 08/03/20212 minutes to read

A rasterizer-variable interface accesses rasterizer state.

Inheritance

The **ID3D10EffectRasterizerVariable** interface inherits from ID3D10EffectVariable. **ID3D10EffectRasterizerVariable** also has these types of members:

Methods

The **ID3D10EffectRasterizerVariable** interface has these methods.

ID3D10EffectRasterizerVariable::GetBackingStore Get a pointer to a variable that contains rasterizer state.
ID3D10EffectRasterizerVariable::GetRasterizerState Get a pointer to a rasterizer interface.

Remarks

An **ID3D10EffectRasterizerVariable 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. For examples, see Two Ways to Get the State in an Effect Variable.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

Effect Interfaces (Direct3D 10)

ID3D10EffectVariable

ID3D10EffectRasterizerVariable::GetBackingStore method (d3d10effect.h)

Article10/13/2021

Get a pointer to a variable that contains rasterizer state.

Syntax

```
C++

HRESULT GetBackingStore(
  [in] UINT                  Index,
  [in] D3D10_RASTERIZER_DESC *pRasterizerDesc
);
```

Parameters

`[in] Index`

Type: **UINT**

Index into an array of rasterizer-state descriptions. If there is only one rasterizer variable in the effect, use 0.

`[in] pRasterizerDesc`

Type: **D3D10_RASTERIZER_DESC***

A pointer to a rasterizer-state description (see D3D10_RASTERIZER_DESC).

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 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 used to

recreate the variable when necessary.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectRasterizerVariable Interface

ID3D10EffectRasterizerVariable::GetRasterizerState method (d3d10effect.h)

Article10/13/2021

Get a pointer to a rasterizer interface.

Syntax

```
C++

HRESULT GetRasterizerState(
  [in]  UINT                  Index,
  [out] ID3D10RasterizerState **ppRasterizerState
);
```

Parameters

`[in] Index`

Type: **UINT**

Index into an array of rasterizer interfaces. If there is only one rasterizer interface, use 0.

`[out] ppRasterizerState`

Type: **ID3D10RasterizerState****

The address of a pointer to a rasterizer interface (see ID3D10RasterizerState Interface).

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows

Header	d3d10effect.h

See also

ID3D10EffectRasterizerVariable Interface

ID3D10EffectRenderTargetViewVariable interface (d3d10effect.h)

Article • 07/22/20212 minutes to read

A render-target-view interface accesses a render target.

Inheritance

The **ID3D10EffectRenderTargetViewVariable** interface inherits from ID3D10EffectVariable. **ID3D10EffectRenderTargetViewVariable** also has these types of members:

Methods

The **ID3D10EffectRenderTargetViewVariable** interface has these methods.

ID3D10EffectRenderTargetViewVariable::GetRenderTarget Get a render-target.
ID3D10EffectRenderTargetViewVariable::GetRenderTargetArray Get an array of render-targets.
ID3D10EffectRenderTargetViewVariable::SetRenderTarget Set a render-target.
ID3D10EffectRenderTargetViewVariable::SetRenderTargetArray Set an array of render-targets.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

Effect Interfaces (Direct3D 10)

ID3D10EffectVariable

ID3D10EffectRenderTargetViewVariable::GetRenderTarget method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get a render-target.

Syntax

```
C++

HRESULT GetRenderTarget(
  [out] ID3D10RenderTargetView **ppResource
);
```

Parameters

`[out] ppResource`

Type: **ID3D10RenderTargetView****

The address of a pointer to a render-target-view interface. See ID3D10RenderTargetView Interface.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h (include D3d10)

See also

ID3D10EffectRenderTargetViewVariable Interface

ID3D10EffectRenderTargetViewVariable::GetRenderTargetArray method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get an array of render-targets.

Syntax

```
C++

HRESULT GetRenderTargetArray(
  [out] ID3D10RenderTargetView **ppResources,
  [in]  UINT                   Offset,
  [in]  UINT                   Count
);
```

Parameters

`[out] ppResources`

Type: **ID3D10RenderTargetView****

A pointer to an array of render-target-view interfaces. See ID3D10RenderTargetView Interface.

`[in] Offset`

Type: **UINT**

The zero-based array index to get the first interface.

`[in] Count`

Type: **UINT**

The number of elements in the array.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h (include D3d10)

See also

ID3D10EffectRenderTargetViewVariable Interface

ID3D10EffectRenderTargetViewVariable::SetRenderTarget method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Set a render-target.

Syntax

C++

```
HRESULT SetRenderTarget(
  [in] ID3D10RenderTargetView *pResource
);
```

Parameters

`[in] pResource`

Type: **ID3D10RenderTargetView***

A pointer to a render-target-view interface. See ID3D10RenderTargetView Interface.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h (include D3d10)

See also

ID3D10EffectRenderTargetViewVariable Interface

ID3D10EffectRenderTargetViewVariable::SetRenderTargetArray method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Set an array of render-targets.

Syntax

```
C++

HRESULT SetRenderTargetArray(
  [out] ID3D10RenderTargetView **ppResources,
  [in]  UINT                   Offset,
  [in]  UINT                   Count
);
```

Parameters

`[out] ppResources`

Type: **ID3D10RenderTargetView****

Set an array of render-target-view interfaces. See ID3D10RenderTargetView Interface.

`[in] Offset`

Type: **UINT**

The zero-based array index to store the first interface.

`[in] Count`

Type: **UINT**

The number of elements in the array.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h (include D3d10)

See also

ID3D10EffectRenderTargetViewVariable Interface

ID3D10EffectSamplerVariable interface (d3d10effect.h)

Article07/22/2021

A sampler interface accesses sampler state.

Inheritance

The **ID3D10EffectSamplerVariable** interface inherits from ID3D10EffectVariable. **ID3D10EffectSamplerVariable** also has these types of members:

Methods

The **ID3D10EffectSamplerVariable** interface has these methods.

ID3D10EffectSamplerVariable::GetBackingStore Get a pointer to a variable that contains sampler state.
ID3D10EffectSamplerVariable::GetSampler Get a pointer to a sampler interface.

Remarks

An **ID3D10EffectSamplerVariable 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. For examples, see Two Ways to Get the State in an Effect Variable.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

Effect Interfaces (Direct3D 10)

ID3D10EffectVariable

ID3D10EffectSamplerVariable::GetBackingStore method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get a pointer to a variable that contains sampler state.

Syntax

```
C++

HRESULT GetBackingStore(
  [in] UINT               Index,
  [in] D3D10_SAMPLER_DESC *pSamplerDesc
);
```

Parameters

`[in] Index`

Type: **UINT**

Index into an array of sampler descriptions. If there is only one sampler variable in the effect, use 0.

`[in] pSamplerDesc`

Type: **D3D10_SAMPLER_DESC***

A pointer to a sampler description (see D3D10_SAMPLER_DESC).

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectSamplerVariable Interface

ID3D10EffectSamplerVariable::GetSampler method (d3d10effect.h)

Article10/13/2021

Get a pointer to a sampler interface.

Syntax

```
C++

HRESULT GetSampler(
  [in]  UINT               Index,
  [out] ID3D10SamplerState **ppSampler
);
```

Parameters

`[in] Index`

Type: **UINT**

Index into an array of sampler interfaces. If there is only one sampler interface, use 0.

`[out] ppSampler`

Type: **ID3D10SamplerState****

The address of a pointer to a sampler interface (see ID3D10SamplerState Interface).

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows

Header	d3d10effect.h

See also

ID3D10EffectSamplerVariable Interface

ID3D10EffectScalarVariable interface (d3d10effect.h)

Article • 07/22/20212 minutes to read

An effect-scalar-variable interface accesses scalar values.

Inheritance

The **ID3D10EffectScalarVariable** interface inherits from ID3D10EffectVariable. **ID3D10EffectScalarVariable** also has these types of members:

Methods

The **ID3D10EffectScalarVariable** interface has these methods.

ID3D10EffectScalarVariable::GetBool Get a boolean variable.
ID3D10EffectScalarVariable::GetBoolArray Get an array of boolean variables.
ID3D10EffectScalarVariable::GetFloat Get a floating-point variable.
ID3D10EffectScalarVariable::GetFloatArray Get an array of floating-point variables.
ID3D10EffectScalarVariable::GetInt Get an integer variable.
ID3D10EffectScalarVariable::GetIntArray Get an array of integer variables.
ID3D10EffectScalarVariable::SetBool Set a boolean variable.

ID3D10EffectScalarVariable::SetBoolArray Set an array of boolean variables.	
ID3D10EffectScalarVariable::SetFloat Set a floating-point variable.	
ID3D10EffectScalarVariable::SetFloatArray Set an array of floating-point variables.	
ID3D10EffectScalarVariable::SetInt Set an integer variable.	
ID3D10EffectScalarVariable::SetIntArray Set an array of integer variables.	

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

Effect Interfaces (Direct3D 10)

ID3D10EffectVariable

ID3D10EffectScalarVariable::GetBool method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get a boolean variable.

Syntax

```
C++

HRESULT GetBool(
  [out] BOOL *pValue
);
```

Parameters

`[out] pValue`

Type: **BOOL***

A pointer to the variable.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectScalarVariable Interface

ID3D10EffectScalarVariable::GetBoolArray method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get an array of boolean variables.

Syntax

```
C++

HRESULT GetBoolArray(
  [out] BOOL *pData,
  [in]  UINT Offset,
  [in]  UINT Count
);
```

Parameters

`[out] pData`

Type: **BOOL***

A pointer to the start of the data to set.

`[in] Offset`

Type: **UINT**

Must be set to 0; this is reserved for future use.

`[in] Count`

Type: **UINT**

The number of array elements to set.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectScalarVariable Interface

ID3D10EffectScalarVariable::GetFloat method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get a floating-point variable.

Syntax

```
C++

HRESULT GetFloat(
  [out] float *pValue
);
```

Parameters

`[out] pValue`

Type: **float***

A pointer to the variable.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectScalarVariable Interface

ID3D10EffectScalarVariable::GetFloatArray method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get an array of floating-point variables.

Syntax

C++

```
HRESULT GetFloatArray(
  [out] float *pData,
  [in]  UINT  Offset,
  [in]  UINT  Count
);
```

Parameters

`[out] pData`

Type: **float***

A pointer to the start of the data to set.

`[in] Offset`

Type: **UINT**

Must be set to 0; this is reserved for future use.

`[in] Count`

Type: **UINT**

The number of array elements to set.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectScalarVariable Interface

ID3D10EffectScalarVariable::GetInt method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get an integer variable.

Syntax

```
C++

HRESULT GetInt(
  [out] int *pValue
);
```

Parameters

`[out] pValue`

Type: **int***

A pointer to the variable.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectScalarVariable Interface

ID3D10EffectScalarVariable::GetIntArray method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get an array of integer variables.

Syntax

```
C++

HRESULT GetIntArray(
  [out] int  *pData,
  [in]  UINT Offset,
  [in]  UINT Count
);
```

Parameters

`[out] pData`

Type: **int***

A pointer to the start of the data to set.

`[in] Offset`

Type: **UINT**

Must be set to 0; this is reserved for future use.

`[in] Count`

Type: **UINT**

The number of array elements to set.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectScalarVariable Interface

ID3D10EffectScalarVariable::SetBool method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Set a boolean variable.

Syntax

```
C++

HRESULT SetBool(
  [in] BOOL Value
);
```

Parameters

`[in] Value`

Type: **BOOL**

A pointer to the variable.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectScalarVariable Interface

ID3D10EffectScalarVariable::SetBoolArray method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Set an array of boolean variables.

Syntax

```
C++

HRESULT SetBoolArray(
  [in] BOOL *pData,
  [in] UINT Offset,
  [in] UINT Count
);
```

Parameters

`[in] pData`

Type: **BOOL***

A pointer to the start of the data to set.

`[in] Offset`

Type: **UINT**

Must be set to 0; this is reserved for future use.

`[in] Count`

Type: **UINT**

The number of array elements to set.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectScalarVariable Interface

ID3D10EffectScalarVariable::SetFloat method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Set a floating-point variable.

Syntax

```
C++

HRESULT SetFloat(
  [in] float Value
);
```

Parameters

`[in] Value`

Type: **float**

A pointer to the variable.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectScalarVariable Interface

ID3D10EffectScalarVariable::SetFloatArray method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Set an array of floating-point variables.

Syntax

```
C++

HRESULT SetFloatArray(
  [in] float *pData,
  [in] UINT  Offset,
  [in] UINT  Count
);
```

Parameters

`[in] pData`

Type: **float***

A pointer to the start of the data to set.

`[in] Offset`

Type: **UINT**

Must be set to 0; this is reserved for future use.

`[in] Count`

Type: **UINT**

The number of array elements to set.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectScalarVariable Interface

ID3D10EffectScalarVariable::SetInt method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Set an integer variable.

Syntax

```
C++

HRESULT SetInt(
  [in] int Value
);
```

Parameters

`[in] Value`

Type: **int**

A pointer to the variable.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectScalarVariable Interface

ID3D10EffectScalarVariable::SetIntArray method (d3d10effect.h)

Article10/13/2021

Set an array of integer variables.

Syntax

```
C++

HRESULT SetIntArray(
  [in] int  *pData,
  [in] UINT Offset,
  [in] UINT Count
);
```

Parameters

`[in] pData`

Type: **int***

A pointer to the start of the data to set.

`[in] Offset`

Type: **UINT**

Must be set to 0; this is reserved for future use.

`[in] Count`

Type: **UINT**

The number of array elements to set.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectScalarVariable Interface

ID3D10EffectShaderResourceVariable interface (d3d10effect.h)

Article • 07/22/20212 minutes to read

A shader-resource interface accesses a shader resource.

Inheritance

The **ID3D10EffectShaderResourceVariable** interface inherits from ID3D10EffectVariable. **ID3D10EffectShaderResourceVariable** also has these types of members:

Methods

The **ID3D10EffectShaderResourceVariable** interface has these methods.

ID3D10EffectShaderResourceVariable::GetResource Get a shader resource.
ID3D10EffectShaderResourceVariable::GetResourceArray Get an array of shader resources.
ID3D10EffectShaderResourceVariable::SetResource Set a shader resource.
ID3D10EffectShaderResourceVariable::SetResourceArray Set an array of shader resources.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

Effect Interfaces (Direct3D 10)

ID3D10EffectVariable

ID3D10EffectShaderResourceVariable::GetResource method (d3d10effect.h)

Article10/13/2021

Get a shader resource.

Syntax

```
C++

HRESULT GetResource(
  [out] ID3D10ShaderResourceView **ppResource
);
```

Parameters

`[out] ppResource`

Type: **ID3D10ShaderResourceView****

The address of a pointer to a shader-resource-view interface. See ID3D10ShaderResourceView Interface.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectShaderResourceVariable Interface

ID3D10EffectShaderResourceVariable::GetResourceArray method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get an array of shader resources.

Syntax

C++

```
HRESULT GetResourceArray(
  [out] ID3D10ShaderResourceView **ppResources,
  [in]  UINT                     Offset,
  [in]  UINT                     Count
);
```

Parameters

`[out] ppResources`

Type: **ID3D10ShaderResourceView****

The address of an array of shader-resource-view interfaces. See ID3D10ShaderResourceView Interface.

`[in] Offset`

Type: **UINT**

The zero-based array index to get the first interface.

`[in] Count`

Type: **UINT**

The number of elements in the array.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectShaderResourceVariable Interface

ID3D10EffectShaderResourceVariable::SetResource method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Set a shader resource.

Syntax

```
C++

HRESULT SetResource(
  [in] ID3D10ShaderResourceView *pResource
);
```

Parameters

`[in] pResource`

Type: **ID3D10ShaderResourceView***

The address of a pointer to a shader-resource-view interface. See ID3D10ShaderResourceView Interface.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectShaderResourceVariable Interface

ID3D10EffectShaderResourceVariable::SetResourceArray method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Set an array of shader resources.

Syntax

C++

```
HRESULT SetResourceArray(
  [out] ID3D10ShaderResourceView **ppResources,
  [in]  UINT                     Offset,
  [in]  UINT                     Count
);
```

Parameters

`[out] ppResources`

Type: **ID3D10ShaderResourceView****

The address of an array of shader-resource-view interfaces. See ID3D10ShaderResourceView Interface.

`[in] Offset`

Type: **UINT**

The zero-based array index to get the first interface.

`[in] Count`

Type: **UINT**

The number of elements in the array.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectShaderResourceVariable Interface

ID3D10EffectShaderVariable interface (d3d10effect.h)

Article • 07/27/20222 minutes to read

A shader-variable interface accesses a shader variable.

Inheritance

The **ID3D10EffectShaderVariable** interface inherits from ID3D10EffectVariable. **ID3D10EffectShaderVariable** also has these types of members:

Methods

The **ID3D10EffectShaderVariable** interface has these methods.

ID3D10EffectShaderVariable::GetGeometryShader Get a geometry shader.
ID3D10EffectShaderVariable::GetInputSignatureElementDesc Get an input-signature description.
ID3D10EffectShaderVariable::GetOutputSignatureElementDesc Get an output-signature description.
ID3D10EffectShaderVariable::GetPixelShader Get a pixel shader.
ID3D10EffectShaderVariable::GetShaderDesc Get a shader description. (ID3D10EffectShaderVariable.GetShaderDesc)
ID3D10EffectShaderVariable::GetVertexShader Get a vertex shader.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

Effect Interfaces (Direct3D 10)

ID3D10EffectVariable

ID3D10EffectShaderVariable::GetGeometryShader method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get a geometry shader.

Syntax

C++

```
HRESULT GetGeometryShader(
  [in]  UINT                 ShaderIndex,
  [out] ID3D10GeometryShader **ppGS
);
```

Parameters

`[in] ShaderIndex`

Type: **UINT**

A zero-based index.

`[out] ppGS`

Type: **ID3D10GeometryShader****

A pointer to a ID3D10GeometryShader Interface.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows

Header	d3d10effect.h

See also

ID3D10EffectShaderResourceVariable Interface

ID3D10EffectShaderVariable

ID3D10EffectShaderVariable::GetInputSignatureElementDesc method (d3d10effect.h)

Article10/13/2021

Get an input-signature description.

Syntax

```
C++

HRESULT GetInputSignatureElementDesc(
  [in] UINT                           ShaderIndex,
  [in] UINT                           Element,
  [in] D3D10_SIGNATURE_PARAMETER_DESC *pDesc
);
```

Parameters

`[in] ShaderIndex`

Type: **UINT**

A zero-based shader index.

`[in] Element`

Type: **UINT**

A zero-based shader-element index.

`[in] pDesc`

Type: **D3D10_SIGNATURE_PARAMETER_DESC***

A pointer to a parameter description (see D3D10_SIGNATURE_PARAMETER_DESC).

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 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.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectShaderResourceVariable Interface

ID3D10EffectShaderVariable

ID3D10EffectShaderVariable::GetOutputSignatureElementDesc method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get an output-signature description.

Syntax

```
C++

HRESULT GetOutputSignatureElementDesc(
  [in] UINT                          ShaderIndex,
  [in] UINT                          Element,
  [in] D3D10_SIGNATURE_PARAMETER_DESC *pDesc
);
```

Parameters

`[in] ShaderIndex`

Type: **UINT**

A zero-based shader index.

`[in] Element`

Type: **UINT**

A zero-based element index.

`[in] pDesc`

Type: **D3D10_SIGNATURE_PARAMETER_DESC***

A pointer to a parameter description (see D3D10_SIGNATURE_PARAMETER_DESC).

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 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.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectShaderResourceVariable Interface

ID3D10EffectShaderVariable

ID3D10EffectShaderVariable::GetPixelShader method (d3d10effect.h)

Article10/13/2021

Get a pixel shader.

Syntax

```
C++

HRESULT GetPixelShader(
  [in]  UINT              ShaderIndex,
  [out] ID3D10PixelShader **ppPS
);
```

Parameters

`[in] ShaderIndex`

Type: **UINT**

A zero-based index.

`[out] ppPS`

Type: **ID3D10PixelShader****

A pointer to a ID3D10PixelShader Interface.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows

Header	d3d10effect.h

See also

ID3D10EffectShaderResourceVariable Interface

ID3D10EffectShaderVariable

ID3D10EffectShaderVariable::GetShaderDesc method (d3d10effect.h)

Article • 07/27/20222 minutes to read

Get a shader description.

Syntax

C++

```
HRESULT GetShaderDesc(
  [in] UINT                     ShaderIndex,
  [in] D3D10_EFFECT_SHADER_DESC *pDesc
);
```

Parameters

`[in] ShaderIndex`

Type: **UINT**

A zero-based index.

`[in] pDesc`

Type: **D3D10_EFFECT_SHADER_DESC***

A pointer to a shader description (see D3D10_EFFECT_SHADER_DESC).

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows

Header	d3d10effect.h

See also

ID3D10EffectShaderResourceVariable Interface

ID3D10EffectShaderVariable

ID3D10EffectShaderVariable::GetVertexShader method (d3d10effect.h)

Article10/13/2021

Get a vertex shader.

Syntax

```
C++

HRESULT GetVertexShader(
  [in]  UINT                ShaderIndex,
  [out] ID3D10VertexShader **ppVS
);
```

Parameters

`[in] ShaderIndex`

Type: **UINT**

A zero-based index.

`[out] ppVS`

Type: **ID3D10VertexShader****

A pointer to a ID3D10VertexShader Interface.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows

Header	d3d10effect.h

See also

ID3D10EffectShaderResourceVariable Interface

ID3D10EffectShaderVariable

ID3D10EffectStringVariable interface (d3d10effect.h)

Article07/22/2021

A string-variable interface accesses a string variable.

Inheritance

The **ID3D10EffectStringVariable** interface inherits from ID3D10EffectVariable. **ID3D10EffectStringVariable** also has these types of members:

Methods

The **ID3D10EffectStringVariable** interface has these methods.

ID3D10EffectStringVariable::GetString Get the string.
ID3D10EffectStringVariable::GetStringArray Get an array of strings.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

Effect Interfaces (Direct3D 10)

ID3D10EffectVariable

ID3D10EffectStringVariable::GetString method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get the string.

Syntax

```
C++

HRESULT GetString(
  [out] LPCSTR *ppString
);
```

Parameters

`[out] ppString`

Type: **LPCSTR***

A pointer to the string.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectStringVariable Interface

ID3D10EffectStringVariable::GetStringArray method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get an array of strings.

Syntax

```
C++

HRESULT GetStringArray(
  [out] LPCSTR *ppStrings,
  [in]  UINT   Offset,
  [in]  UINT   Count
);
```

Parameters

`[out] ppStrings`

Type: **LPCSTR***

A pointer to the first string in the array.

`[in] Offset`

Type: **UINT**

The offset (in number of strings) between the start of the array and the first string to get.

`[in] Count`

Type: **UINT**

The number of strings in the returned array.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectStringVariable Interface

ID3D10EffectTechnique interface (d3d10effect.h)

Article08/19/2022

An **ID3D10EffectTechnique** interface is a collection of passes.

The lifetime of an **ID3D10EffectTechnique** object is equal to the lifetime of its parent ID3D10Effect object.

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.

Methods

The **ID3D10EffectTechnique** interface has these methods.

ID3D10EffectTechnique::ComputeStateBlockMask Compute a state-block mask to allow/prevent state changes.
ID3D10EffectTechnique::GetAnnotationByIndex The ID3D10EffectTechnique::GetAnnotationByIndex (d3d10effect.h) method gets an annotation by index.
ID3D10EffectTechnique::GetAnnotationByName Get an annotation by name. (ID3D10EffectTechnique.GetAnnotationByName)

ID3D10EffectTechnique::GetDesc Get a technique description.	
ID3D10EffectTechnique::GetPassByIndex Get a pass by index.	
ID3D10EffectTechnique::GetPassByName Get a pass by name.	
ID3D10EffectTechnique::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 (see Organizing State in an Effect (Direct3D 10)). The syntax for creating a technique is shown in Effect Technique Syntax (Direct3D 10).

To get an effect-technique interface, call a method like ID3D10Effect::GetTechniqueByName.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

Effect Interfaces (Direct3D 10)

ID3D10EffectTechnique::ComputeStateBlockMask method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Compute a state-block mask to allow/prevent state changes.

Syntax

```
C++

HRESULT ComputeStateBlockMask(
  [in] D3D10_STATE_BLOCK_MASK *pStateBlockMask
);
```

Parameters

`[in] pStateBlockMask`

Type: **D3D10_STATE_BLOCK_MASK***

A pointer to a state-block mask (see D3D10_STATE_BLOCK_MASK).

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectTechnique Interface

ID3D10EffectTechnique::GetAnnotationByIndex method (d3d10effect.h)

Article • 08/19/20222 minutes to read

Get an annotation by index.

Syntax

```
C++

ID3D10EffectVariable * GetAnnotationByIndex(
  [in] UINT Index
);
```

Parameters

`[in] Index`

Type: **UINT**

The zero-based index of the interface pointer.

Return value

Type: **ID3D10EffectVariable***

A pointer to an ID3D10EffectVariable Interface.

Remarks

Use an annotation to attach a piece of metadata to a technique.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectTechnique Interface

ID3D10EffectTechnique::GetAnnotationByName method (d3d10effect.h)

Article • 07/27/20222 minutes to read

Get an annotation by name.

Syntax

```
C++

ID3D10EffectVariable * GetAnnotationByName(
  [in] LPCSTR Name
);
```

Parameters

`[in] Name`

Type: **LPCSTR**

Name of the annotation.

Return value

Type: **ID3D10EffectVariable***

A pointer to an ID3D10EffectVariable Interface.

Remarks

Use an annotation to attach a piece of metadata to a technique.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectTechnique Interface

ID3D10EffectTechnique::GetDesc method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get a technique description.

Syntax

```
C++

HRESULT GetDesc(
  [in] D3D10_TECHNIQUE_DESC *pDesc
);
```

Parameters

`[in] pDesc`

Type: **D3D10_TECHNIQUE_DESC***

A pointer to a technique description (see D3D10_TECHNIQUE_DESC).

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectTechnique Interface

ID3D10EffectTechnique::GetPassByIndex method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get a pass by index.

Syntax

```
C++

ID3D10EffectPass * GetPassByIndex(
  [in] UINT Index
);
```

Parameters

`[in] Index`

Type: **UINT**

A zero-based index.

Return value

Type: **ID3D10EffectPass***

A pointer to a ID3D10EffectPass Interface.

Remarks

A technique contains one or more passes; get a pass using a name or an index.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectTechnique Interface

ID3D10EffectTechnique::GetPassByName method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get a pass by name.

Syntax

```
C++

ID3D10EffectPass * GetPassByName(
  [in] LPCSTR Name
);
```

Parameters

`[in] Name`

Type: **LPCSTR**

The name of the pass.

Return value

Type: **ID3D10EffectPass***

A pointer to an ID3D10EffectPass Interface.

Remarks

A technique contains one or more passes; get a pass using a name or an index.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectTechnique Interface

ID3D10EffectTechnique::IsValid method (d3d10effect.h)

Article06/29/2021

Test a technique to see if it contains valid syntax.

Syntax

```
C++

BOOL IsValid();
```

Return value

Type: **BOOL**

TRUE if the code syntax is valid; otherwise **FALSE**.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectTechnique Interface

ID3D10EffectType interface (d3d10effect.h)

Article10/06/2021

The **ID3D10EffectType** interface accesses effect variables by type.

The lifetime of an **ID3D10EffectType** object is equal to the lifetime of its parent ID3D10Effect object.

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 a member type by name.
GetMemberTypeBySemantic	Get a member type by semantic.
IsValid	Tests that the effect type is valid.

Methods

The **ID3D10EffectType** interface has these methods.

ID3D10EffectType::GetDesc Get an effect-type description.
ID3D10EffectType::GetMemberName Get the name of a member.
ID3D10EffectType::GetMemberSemantic Get the semantic attached to a member.
ID3D10EffectType::GetMemberTypeByIndex Get a member type by index.

ID3D10EffectType::GetMemberTypeByName Get a member type by name.	
ID3D10EffectType::GetMemberTypeBySemantic Get a member type by semantic.	
ID3D10EffectType::IsValid Tests that the effect type is valid.	

Remarks

To get information about an effect type from an effect variable, call ID3D10EffectVariable::GetType.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

Effect Interfaces (Direct3D 10)

ID3D10EffectType::GetDesc method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get an effect-type description.

Syntax

```
C++

HRESULT GetDesc(
  [in] D3D10_EFFECT_TYPE_DESC *pDesc
);
```

Parameters

`[in] pDesc`

Type: **D3D10_EFFECT_TYPE_DESC***

A pointer to an effect-type description. See D3D10_EFFECT_TYPE_DESC.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Remarks

The effect-variable description contains data about the name, annotations, semantic, flags and buffer offset of the effect type.

Requirements

Target Platform	Windows

Header	d3d10effect.h

See also

ID3D10EffectType Interface

ID3D10EffectType::GetMemberName method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get the name of a member.

Syntax

```
C++

LPCSTR GetMemberName(
  [in] UINT Index
);
```

Parameters

`[in] Index`

Type: **UINT**

A zero-based index.

Return value

Type: **LPCSTR**

The name of the member.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectType Interface

ID3D10EffectType::GetMemberSemantic method (d3d10effect.h)

Article10/13/2021

Get the semantic attached to a member.

Syntax

```
C++

LPCSTR GetMemberSemantic(
  [in] UINT Index
);
```

Parameters

`[in] Index`

Type: **UINT**

A zero-based index.

Return value

Type: **LPCSTR**

A string that contains the semantic.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectType Interface

ID3D10EffectType::GetMemberTypeByIndex method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get a member type by index.

Syntax

```
C++

ID3D10EffectType * GetMemberTypeByIndex(
  [in] UINT Index
);
```

Parameters

`[in] Index`

Type: **UINT**

A zero-based index.

Return value

Type: **ID3D10EffectType***

A pointer to an ID3D10EffectType Interface.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectType Interface

ID3D10EffectType::GetMemberTypeByName method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get a member type by name.

Syntax

```
C++

ID3D10EffectType * GetMemberTypeByName(
  [in] LPCSTR Name
);
```

Parameters

`[in] Name`

Type: **LPCSTR**

A member's name.

Return value

Type: **ID3D10EffectType***

A pointer to an ID3D10EffectType Interface.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectType Interface

ID3D10EffectType::GetMemberTypeBySemantic method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get a member type by semantic.

Syntax

```
C++

ID3D10EffectType * GetMemberTypeBySemantic(
  [in] LPCSTR Semantic
);
```

Parameters

`[in] Semantic`

Type: **LPCSTR**

A semantic.

Return value

Type: **ID3D10EffectType***

A pointer to an ID3D10EffectType Interface.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectType Interface

ID3D10EffectType::IsValid method (d3d10effect.h)

Article • 06/29/20212 minutes to read

Tests that the effect type is valid.

Syntax

```
C++

BOOL IsValid();
```

Return value

Type: **BOOL**

TRUE if it is valid; otherwise **FALSE**.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectType Interface

ID3D10EffectVariable interface (d3d10effect.h)

Article • 08/19/20222 minutes to read

The **ID3D10EffectVariable** interface is the base class for all effect variables.

The lifetime of an **ID3D10EffectVariable** object is equal to the lifetime of its parent ID3D10Effect object.

Method	Description
AsBlend	Get a effect-blend variable.
AsConstantBuffer	Get a constant buffer.
AsDepthStencil	Get a depth-stencil variable.
AsDepthStencilView	Get a depth-stencil-view 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.
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.
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.

Methods

The **ID3D10EffectVariable** interface has these methods.

ID3D10EffectVariable::AsBlend Get a effect-blend variable.
ID3D10EffectVariable::AsConstantBuffer Get a constant buffer. (ID3D10EffectVariable.AsConstantBuffer)
ID3D10EffectVariable::AsDepthStencil Get a depth-stencil variable.
ID3D10EffectVariable::AsDepthStencilView Get a depth-stencil-view variable.
ID3D10EffectVariable::AsMatrix Get a matrix variable.
ID3D10EffectVariable::AsRasterizer Get a rasterizer variable.
ID3D10EffectVariable::AsRenderTargetView Get a render-target-view variable.
ID3D10EffectVariable::AsSampler Get a sampler variable.

ID3D10EffectVariable::AsScalar	Get a scalar variable.
ID3D10EffectVariable::AsShader	Get a shader variable.
ID3D10EffectVariable::AsShaderResource	Get a shader-resource variable.
ID3D10EffectVariable::AsString	Get a string variable.
ID3D10EffectVariable::AsVector	Get a vector variable.
ID3D10EffectVariable::GetAnnotationByIndex	The ID3D10EffectVariable::GetAnnotationByIndex (d3d10effect.h) method gets an annotation by index.
ID3D10EffectVariable::GetAnnotationByName	Get an annotation by name. (ID3D10EffectVariable.GetAnnotationByName)
ID3D10EffectVariable::GetDesc	Get a description.
ID3D10EffectVariable::GetElement	Get an array element.
ID3D10EffectVariable::GetMemberByIndex	Get a structure member by index.
ID3D10EffectVariable::GetMemberByName	Get a structure member by name.
ID3D10EffectVariable::GetMemberBySemantic	Get a structure member by semantic.

ID3D10EffectVariable::GetParentConstantBuffer Get a constant buffer. (ID3D10EffectVariable.GetParentConstantBuffer)	
ID3D10EffectVariable::GetRawValue Get data.	
ID3D10EffectVariable::GetType Get type information.	
ID3D10EffectVariable::IsValid Compare the data type with the data stored.	
ID3D10EffectVariable::SetRawValue Set data.	

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

Effect Interfaces (Direct3D 10)

ID3D10EffectVariable::AsBlend method (d3d10effect.h)

Article • 06/29/20212 minutes to read

Get a effect-blend variable.

Syntax

```
C++

ID3D10EffectBlendVariable * AsBlend();
```

Return value

Type: **ID3D10EffectBlendVariable***

A pointer to an effect blend variable. See ID3D10EffectBlendVariable.

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.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectVariable Interface

ID3D10EffectVariable::AsConstantBuffer method (d3d10effect.h)

Article • 07/27/20222 minutes to read

Get a constant buffer.

Syntax

```
C++

ID3D10EffectConstantBuffer * AsConstantBuffer();
```

Return value

Type: **ID3D10EffectConstantBuffer***

A pointer to a constant buffer. See ID3D10EffectConstantBuffer.

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.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectVariable Interface

ID3D10EffectVariable::AsDepthStencil method (d3d10effect.h)

Article • 06/29/20212 minutes to read

Get a depth-stencil variable.

Syntax

```
C++

ID3D10EffectDepthStencilVariable * AsDepthStencil();
```

Return value

Type: **ID3D10EffectDepthStencilVariable***

A pointer to a depth-stencil variable. See ID3D10EffectDepthStencilVariable.

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.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectVariable Interface

ID3D10EffectVariable::AsDepthStencilView method (d3d10effect.h)

Article06/29/2021

Get a depth-stencil-view variable.

Syntax

```
C++

ID3D10EffectDepthStencilViewVariable * AsDepthStencilView();
```

Return value

Type: **ID3D10EffectDepthStencilViewVariable***

A pointer to a depth-stencil-view variable. See ID3D10EffectDepthStencilViewVariable Interface.

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.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectVariable Interface

ID3D10EffectVariable::AsMatrix method (d3d10effect.h)

Article • 06/29/20212 minutes to read

Get a matrix variable.

Syntax

```
C++

ID3D10EffectMatrixVariable * AsMatrix();
```

Return value

Type: **ID3D10EffectMatrixVariable***

A pointer to a matrix variable. See ID3D10EffectMatrixVariable.

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.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectVariable Interface

ID3D10EffectVariable::AsRasterizer method (d3d10effect.h)

Article • 06/29/20212 minutes to read

Get a rasterizer variable.

Syntax

```
C++

ID3D10EffectRasterizerVariable * AsRasterizer();
```

Return value

Type: **ID3D10EffectRasterizerVariable***

A pointer to a rasterizer variable. See ID3D10EffectRasterizerVariable.

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.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectVariable Interface

ID3D10EffectVariable::AsRenderTargetView method (d3d10effect.h)

Article06/29/2021

Get a render-target-view variable.

Syntax

```
C++

ID3D10EffectRenderTargetViewVariable * AsRenderTargetView();
```

Return value

Type: **ID3D10EffectRenderTargetViewVariable***

A pointer to a render-target-view variable. See ID3D10EffectRenderTargetViewVariable Interface.

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.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectVariable Interface

ID3D10EffectVariable::AsSampler method (d3d10effect.h)

Article06/29/2021

Get a sampler variable.

Syntax

```
C++

ID3D10EffectSamplerVariable * AsSampler();
```

Return value

Type: **ID3D10EffectSamplerVariable***

A pointer to a sampler variable. See ID3D10EffectSamplerVariable.

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.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectVariable Interface

ID3D10EffectVariable::AsScalar method (d3d10effect.h)

Article • 06/29/20212 minutes to read

Get a scalar variable.

Syntax

```
C++

ID3D10EffectScalarVariable * AsScalar();
```

Return value

Type: **ID3D10EffectScalarVariable***

A pointer to a scalar variable. See ID3D10EffectScalarVariable.

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.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectVariable Interface

ID3D10EffectVariable::AsShader method (d3d10effect.h)

Article06/29/2021

Get a shader variable.

Syntax

```
C++

ID3D10EffectShaderVariable * AsShader();
```

Return value

Type: **ID3D10EffectShaderVariable***

A pointer to a shader variable. See ID3D10EffectShaderVariable.

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.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectVariable Interface

ID3D10EffectVariable::AsShaderResource method (d3d10effect.h)

Article06/29/2021

Get a shader-resource variable.

Syntax

```
C++

ID3D10EffectShaderResourceVariable * AsShaderResource();
```

Return value

Type: **ID3D10EffectShaderResourceVariable***

A pointer to a shader-resource variable. See ID3D10EffectShaderResourceVariable.

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.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectVariable Interface

ID3D10EffectVariable::AsString method (d3d10effect.h)

Article • 06/29/20212 minutes to read

Get a string variable.

Syntax

```
C++

ID3D10EffectStringVariable * AsString();
```

Return value

Type: **ID3D10EffectStringVariable***

A pointer to a string variable. See ID3D10EffectStringVariable.

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.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectVariable Interface

ID3D10EffectVariable::AsVector method (d3d10effect.h)

Article • 06/29/20212 minutes to read

Get a vector variable.

Syntax

```
C++

ID3D10EffectVectorVariable * AsVector();
```

Return value

Type: **ID3D10EffectVectorVariable***

A pointer to a vector variable. See ID3D10EffectVectorVariable.

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.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectVariable Interface

ID3D10EffectVariable::GetAnnotationByIndex method (d3d10effect.h)

Article • 08/19/20222 minutes to read

Get an annotation by index.

Syntax

```
C++

ID3D10EffectVariable * GetAnnotationByIndex(
  [in] UINT Index
);
```

Parameters

`[in] Index`

Type: **UINT**

A zero-based index.

Return value

Type: **ID3D10EffectVariable***

A pointer to an ID3D10EffectVariable Interface.

Remarks

Annotations can be attached to a technique, a pass or a global variable. For the syntax, see Annotation Syntax (Direct3D 10).

Requirements

Target Platform	Windows

Header	d3d10effect.h

See also

ID3D10EffectVariable Interface

ID3D10EffectVariable::GetAnnotationByName method (d3d10effect.h)

Article • 07/27/20222 minutes to read

Get an annotation by name.

Syntax

```
C++

ID3D10EffectVariable * GetAnnotationByName(
  [in] LPCSTR Name
);
```

Parameters

`[in] Name`

Type: **LPCSTR**

The annotation name.

Return value

Type: **ID3D10EffectVariable***

A pointer to an ID3D10EffectVariable Interface. Note that if the annotation is not found the **ID3D10EffectVariable Interface** returned will be empty. The ID3D10EffectVariable::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. For the syntax, see Annotation Syntax (Direct3D 10).

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectVariable Interface

ID3D10EffectVariable::GetDesc method (d3d10effect.h)

Article10/13/2021

Get a description.

Syntax

```
C++

HRESULT GetDesc(
  [in] D3D10_EFFECT_VARIABLE_DESC *pDesc
);
```

Parameters

`[in] pDesc`

Type: **D3D10_EFFECT_VARIABLE_DESC***

A pointer to an effect-variable description (see D3D10_EFFECT_VARIABLE_DESC).

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectVariable Interface

ID3D10EffectVariable::GetElement method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get an array element.

Syntax

```
C++

ID3D10EffectVariable * GetElement(
  [in] UINT Index
);
```

Parameters

`[in] Index`

Type: **UINT**

A zero-based index; otherwise 0.

Return value

Type: **ID3D10EffectVariable***

A pointer to an ID3D10EffectVariable Interface.

Remarks

If the effect variable is an array, use this method to return one of the elements.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectVariable Interface

ID3D10EffectVariable::GetMemberByIndex method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get a structure member by index.

Syntax

```
C++

ID3D10EffectVariable * GetMemberByIndex(
  [in] UINT Index
);
```

Parameters

`[in] Index`

Type: **UINT**

A zero-based index.

Return value

Type: **ID3D10EffectVariable***

A pointer to an ID3D10EffectVariable Interface.

Remarks

If the effect variable is a structure, use this method to look up a member by index.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectVariable Interface

ID3D10EffectVariable::GetMemberByName method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get a structure member by name.

Syntax

```
C++

ID3D10EffectVariable * GetMemberByName(
  [in] LPCSTR Name
);
```

Parameters

`[in] Name`

Type: **LPCSTR**

Member name.

Return value

Type: **ID3D10EffectVariable***

A pointer to an ID3D10EffectVariable Interface.

Remarks

If the effect variable is a structure, use this method to look up a member by name.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectVariable Interface

ID3D10EffectVariable::GetMemberBySemantic method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get a structure member by semantic.

Syntax

```
C++

ID3D10EffectVariable * GetMemberBySemantic(
  [in] LPCSTR Semantic
);
```

Parameters

`[in] Semantic`

Type: **LPCSTR**

The semantic.

Return value

Type: **ID3D10EffectVariable***

A pointer to an ID3D10EffectVariable Interface.

Remarks

If the effect variable is a structure, use this method to look up a member by attached semantic.

Requirements

Target Platform	Windows

Header	d3d10effect.h

See also

ID3D10EffectVariable Interface

ID3D10EffectVariable::GetParentConstantBuffer method (d3d10effect.h)

Article07/27/2022

Get a constant buffer.

Syntax

```
C++

ID3D10EffectConstantBuffer * GetParentConstantBuffer();
```

Return value

Type: **ID3D10EffectConstantBuffer***

A pointer to a ID3D10EffectConstantBuffer Interface.

Remarks

Effect variables are read-from or written-to a constant buffer.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectVariable Interface

ID3D10EffectVariable::GetRawValue method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get data.

Syntax

```
C++

HRESULT GetRawValue(
  [in] void *pData,
  [in] UINT Offset,
  [in] UINT ByteCount
);
```

Parameters

`[in] pData`

Type: **void***

A pointer to the variable.

`[in] Offset`

Type: **UINT**

The offset (in bytes) from the beginning of the pointer to the data.

`[in] ByteCount`

Type: **UINT**

The number of bytes to get.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Remarks

This method does no conversion or type checking; it is therefore a very quick way to access array items.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectVariable Interface

ID3D10EffectVariable::GetType method (d3d10effect.h)

Article06/29/2021

Get type information.

Syntax

```
C++

ID3D10EffectType * GetType();
```

Return value

Type: **ID3D10EffectType***

A pointer to an ID3D10EffectType Interface.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectVariable Interface

ID3D10EffectVariable::IsValid method (d3d10effect.h)

Article06/29/2021

Compare the data type with the data stored.

Syntax

```
C++

BOOL IsValid();
```

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).

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectVariable Interface

ID3D10EffectVariable::SetRawValue method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Set data.

Syntax

```
C++

HRESULT SetRawValue(
  [in] void *pData,
  [in] UINT Offset,
  [in] UINT ByteCount
);
```

Parameters

`[in] pData`

Type: **void***

A pointer to the variable.

`[in] Offset`

Type: **UINT**

The offset (in bytes) from the beginning of the pointer to the data.

`[in] ByteCount`

Type: **UINT**

The number of bytes to set.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Remarks

This method does no conversion or type checking; it is therefore a very quick way to access array items.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectVariable Interface

ID3D10EffectVectorVariable interface (d3d10effect.h)

Article • 07/22/20212 minutes to read

A vector-variable interface accesses a four-component vector.

Inheritance

The **ID3D10EffectVectorVariable** interface inherits from ID3D10EffectVariable. **ID3D10EffectVectorVariable** also has these types of members:

Methods

The **ID3D10EffectVectorVariable** interface has these methods.

ID3D10EffectVectorVariable::GetBoolVector Get a four-component vector that contains boolean data.
ID3D10EffectVectorVariable::GetBoolVectorArray Get an array of four-component vectors that contain boolean data.
ID3D10EffectVectorVariable::GetFloatVector Get a four-component vector that contains floating-point data.
ID3D10EffectVectorVariable::GetFloatVectorArray Get an array of four-component vectors that contain floating-point data.
ID3D10EffectVectorVariable::GetIntVector Get a four-component vector that contains integer data.
ID3D10EffectVectorVariable::GetIntVectorArray Get an array of four-component vectors that contain integer data.
ID3D10EffectVectorVariable::SetBoolVector Set a four-component vector that contains boolean data.

ID3D10EffectVectorVariable::SetBoolVectorArray Set an array of four-component vectors that contain boolean data.
ID3D10EffectVectorVariable::SetFloatVector Set a four-component vector that contains floating-point data.
ID3D10EffectVectorVariable::SetFloatVectorArray Set an array of four-component vectors that contain floating-point data.
ID3D10EffectVectorVariable::SetIntVector Set a four-component vector that contains integer data.
ID3D10EffectVectorVariable::SetIntVectorArray Set an array of four-component vectors that contain integer data.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

Effect Interfaces (Direct3D 10)

ID3D10EffectVariable

ID3D10EffectVectorVariable::GetBoolVector method (d3d10effect.h)

Article10/13/2021

Get a four-component vector that contains boolean data.

Syntax

```
C++

HRESULT GetBoolVector(
  [in] BOOL *pData
);
```

Parameters

`[in] pData`

Type: **BOOL***

A pointer to the first component.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectVectorVariable Interface

ID3D10EffectVectorVariable::GetBoolVectorArray method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get an array of four-component vectors that contain boolean data.

Syntax

```
C++

HRESULT GetBoolVectorArray(
  [in] BOOL *pData,
  [in] UINT Offset,
  [in] UINT Count
);
```

Parameters

`[in] pData`

Type: **BOOL***

A pointer to the start of the data to set.

`[in] Offset`

Type: **UINT**

Must be set to 0; this is reserved for future use.

`[in] Count`

Type: **UINT**

The number of array elements to set.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectVectorVariable Interface

ID3D10EffectVectorVariable::GetFloatVector method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get a four-component vector that contains floating-point data.

Syntax

```
C++

HRESULT GetFloatVector(
  [in] float *pData
);
```

Parameters

`[in] pData`

Type: **float***

A pointer to the first component.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectVectorVariable Interface

ID3D10EffectVectorVariable::GetFloatVectorArray method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get an array of four-component vectors that contain floating-point data.

Syntax

```
C++

HRESULT GetFloatVectorArray(
  [in] float *pData,
  [in] UINT  Offset,
  [in] UINT  Count
);
```

Parameters

`[in] pData`

Type: **float***

A pointer to the start of the data to set.

`[in] Offset`

Type: **UINT**

Must be set to 0; this is reserved for future use.

`[in] Count`

Type: **UINT**

The number of array elements to set.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectVectorVariable Interface

ID3D10EffectVectorVariable::GetIntVector method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get a four-component vector that contains integer data.

Syntax

```
C++

HRESULT GetIntVector(
  [in] int *pData
);
```

Parameters

`[in] pData`

Type: **int***

A pointer to the first component.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectVectorVariable Interface

ID3D10EffectVectorVariable::GetIntVectorArray method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get an array of four-component vectors that contain integer data.

Syntax

```
C++

HRESULT GetIntVectorArray(
  [in] int  *pData,
  [in] UINT Offset,
  [in] UINT Count
);
```

Parameters

`[in] pData`

Type: **int***

A pointer to the start of the data to set.

`[in] Offset`

Type: **UINT**

Must be set to 0; this is reserved for future use.

`[in] Count`

Type: **UINT**

The number of array elements to set.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectVectorVariable Interface

ID3D10EffectVectorVariable::SetBoolVector method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Set a four-component vector that contains boolean data.

Syntax

```
C++

HRESULT SetBoolVector(
  [in] BOOL *pData
);
```

Parameters

`[in] pData`

Type: **BOOL***

A pointer to the first component.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectVectorVariable Interface

ID3D10EffectVectorVariable::SetBoolVectorArray method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Set an array of four-component vectors that contain boolean data.

Syntax

```
C++

HRESULT SetBoolVectorArray(
  [in] BOOL *pData,
  [in] UINT Offset,
  [in] UINT Count
);
```

Parameters

`[in] pData`

Type: **BOOL***

A pointer to the start of the data to set.

`[in] Offset`

Type: **UINT**

Must be set to 0; this is reserved for future use.

`[in] Count`

Type: **UINT**

The number of array elements to set.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectVectorVariable Interface

ID3D10EffectVectorVariable::SetFloatVector method (d3d10effect.h)

Article10/13/2021

Set a four-component vector that contains floating-point data.

Syntax

```
C++

HRESULT SetFloatVector(
  [in] float *pData
);
```

Parameters

`[in] pData`

Type: **float***

A pointer to the first component.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectVectorVariable Interface

ID3D10EffectVectorVariable::SetFloatVectorArray method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Set an array of four-component vectors that contain floating-point data.

Syntax

```
C++

HRESULT SetFloatVectorArray(
  [in] float *pData,
  [in] UINT  Offset,
  [in] UINT  Count
);
```

Parameters

`[in] pData`

Type: **float***

A pointer to the start of the data to set.

`[in] Offset`

Type: **UINT**

Must be set to 0; this is reserved for future use.

`[in] Count`

Type: **UINT**

The number of array elements to set.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectVectorVariable Interface

ID3D10EffectVectorVariable::SetIntVector method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Set a four-component vector that contains integer data.

Syntax

```
C++

HRESULT SetIntVector(
  [in] int *pData
);
```

Parameters

`[in] pData`

Type: **int***

A pointer to the first component.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectVectorVariable Interface

ID3D10EffectVectorVariable::SetIntVectorArray method (d3d10effect.h)

Article • 10/13/20212 minutes to read

Set an array of four-component vectors that contain integer data.

Syntax

C++

```
HRESULT SetIntVectorArray(
  [in] int  *pData,
  [in] UINT Offset,
  [in] UINT Count
);
```

Parameters

`[in] pData`

Type: **int***

A pointer to the start of the data to set.

`[in] Offset`

Type: **UINT**

Must be set to 0; this is reserved for future use.

`[in] Count`

Type: **UINT**

The number of array elements to set.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h

See also

ID3D10EffectVectorVariable Interface

Effect Functions (Direct3D 10)

Article • 01/06/2021 • 2 minutes to read

This section contains information about the following effect functions:

Functions	Description
D3D10CompileEffectFromMemory	Compile an effect.
D3D10CreateEffectFromMemory	Create an effect.
D3D10CreateEffectPoolFromMemory	Create an effect pool.
D3D10CreateStateBlock	Create a state block.
D3D10DisassembleEffect	Disassemble a compiled effect into shader assembly instructions.
D3D10StateBlockMaskDifference	Combine two state-block masks with a bitwise XOR.
D3D10StateBlockMaskDisableAll	Disable state capturing.
D3D10StateBlockMaskDisableCapture	Disable state capturing with a state-block mask.
D3D10StateBlockMaskEnableAll	Enable a state-block mask to capture and apply all state variables.
D3D10StateBlockMaskEnableCapture	Enable a range of state values in a state block mask.
D3D10StateBlockMaskGetSetting	Get an element in a state-block mask.
D3D10StateBlockMaskIntersect	Combine two state-block masks with a bitwise AND.
D3D10StateBlockMaskUnion	Combine two state-block masks with a bitwise OR.

Related topics

Effect Reference

D3D10CompileEffectFromMemory function (d3d10effect.h)

Article • 10/13/20212 minutes to read

Compile an effect.

> **Note** Use **D3DX10CompileFromMemory** instead of this function.

Syntax

```C++
HRESULT D3D10CompileEffectFromMemory(
  [in]  void                     *pData,
  [in]  SIZE_T                   DataLength,
  [in]  LPCSTR                   pSrcFileName,
  [in]  const D3D10_SHADER_MACRO *pDefines,
  [in]  ID3D10Include            *pInclude,
  [in]  UINT                     HLSLFlags,
  [in]  UINT                     FXFlags,
  [out] ID3D10Blob               **ppCompiledEffect,
  [out] ID3D10Blob               **ppErrors
);
```

Parameters

`[in] pData`

Type: **void***

A pointer to effect data; either ASCII HLSL code or a compiled effect.

`[in] DataLength`

Type: **SIZE_T**

Length of *pData*.

`[in] pSrcFileName`

Type: **LPCSTR**

The name of the effect file.

`[in] pDefines`

Type: **const D3D10_SHADER_MACRO***

Optional. An array of NULL-terminated macro definitions (see D3D10_SHADER_MACRO).

`[in] pInclude`

Type: **ID3D10Include***

Optional. A pointer to an ID3D10Include Interface for handling include files. Setting this to **NULL** will cause a compile error if a shader contains a #include.

`[in] HLSLFlags`

Type: **UINT**

Shader compile options.

`[in] FXFlags`

Type: **UINT**

Effect compile options.

`[out] ppCompiledEffect`

Type: **ID3D10Blob****

The address of a ID3D10Blob Interface that contains the compiled effect.

`[out] ppErrors`

Type: **ID3D10Blob****

Optional. A pointer to an ID3D10Blob Interface that contains compiler error messages, or **NULL** if there were no errors.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Remarks

This function uses the version of the HLSL compiler released in the November 2006 DirectX SDK.

For an example, see Compile an Effect (Direct3D 10).

Requirements

Target Platform	Windows
Header	d3d10effect.h
Library	D3D10.lib
DLL	D3D10.dll

See also

Effect Functions (Direct3D 10)

D3D10CreateEffectFromMemory function (d3d10effect.h)

Article05/17/2023

Creates an ID3D10Effect from a buffer containing a compiled effect.

Syntax

```
C++

HRESULT D3D10CreateEffectFromMemory(
  [in]  void             *pData,
  [in]  SIZE_T           DataLength,
  [in]  UINT             FXFlags,
  [in]  ID3D10Device     *pDevice,
  [in]  ID3D10EffectPool *pEffectPool,
  [out] ID3D10Effect     **ppEffect
);
```

Parameters

`[in] pData`

Type: **void***

A pointer to a compiled effect.

`[in] DataLength`

Type: **SIZE_T**

Length of *pData*.

`[in] FXFlags`

Type: **UINT**

Effect compile options.

`[in] pDevice`

Type: **ID3D10Device***

A pointer to the device (see ID3D10Device Interface).

`[in] pEffectPool`

Type: **ID3D10EffectPool***

Optional. A pointer to a memory space for effect variables that are shared across effects (see ID3D10EffectPool Interface).

`[out] ppEffect`

Type: **ID3D10Effect****

A pointer to an ID3D10Effect Interface which contains the created effect.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Remarks

> **Note**
>
> Linking `d3d10_1.lib` gives you the implementation in `d3d10_1.dll`, which is the Direct3D10.1 programming model implementation. Linking `d3d10.lib` gives you the implementation in `d3d10.dll`, which is the Direct3D10 programming model implementation.

This method is used to create an ID3D10Effect Interface object from an effect that has been compiled before runtime and loaded into memory. For help precompiling an effect, see Offline Compiling. To load and compile an ASCII .fx file see Compile an Effect (Direct3D 10).

Examples

Compile the effect

```
fxc.exe /T fx_4_0 /Fo Tutorial03.fxo Tutorial03.fx
```

Load the compiled effect at runtime.

```
ifstream is("tutorial03.fxo", ios::binary);
is.seekg(0,ios_base::end);
streampos pos = is.tellg();
is.seekg(0,ios_base::beg);
char * effectBuffer = new char[pos];
is.read(effectBuffer,pos);

hr = D3D10CreateEffectFromMemory((void
*)effectBuffer,pos,0,g_pd3dDevice,NULL,&g_pEffect);
```

Requirements

Target Platform	Windows
Header	d3d10effect.h
Library	d3d10_1.lib, d3d10.lib
DLL	d3d10_1.dll, d3d10.dll

See also

Effect Functions (Direct3D 10)

D3D10CreateEffectPoolFromMemory function (d3d10effect.h)

Article05/17/2023

Create an effect pool (or shared memory location), to enable sharing variables between effects.

Syntax

```
C++

HRESULT D3D10CreateEffectPoolFromMemory(
  [in]  void             *pData,
  [in]  SIZE_T           DataLength,
  [in]  UINT             FXFlags,
  [in]  ID3D10Device     *pDevice,
  [out] ID3D10EffectPool **ppEffectPool
);
```

Parameters

`[in] pData`

Type: **void***

A pointer to a compiled effect.

`[in] DataLength`

Type: **SIZE_T**

Length of *pData*.

`[in] FXFlags`

Type: **UINT**

Effect compile options.

`[in] pDevice`

Type: **ID3D10Device***

A pointer to the device (see ID3D10Device Interface).

`[out] ppEffectPool`

Type: **ID3D10EffectPool****

A pointer to the ID3D10EffectPool Interface that contains the effect pool.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Remarks

> **Note**
>
> Linking `d3d10_1.lib` gives you the implementation in `d3d10_1.dll`, which is the Direct3D10.1 programming model implementation. Linking `d3d10.lib` gives you the implementation in `d3d10.dll`, which is the Direct3D10 programming model implementation.

A pool is a shared location in memory. Effect variables that are located in a pool can be updated once, and the effect system will take care of updating each effect that uses that variable. To pool an effect variable, tell the effect to locate the variable in a pool when the effect is created, using a helper function such as D3DX10CreateEffectFromFile.

For help compiling an effect, see Compile an Effect (Direct3D 10).

Requirements

Target Platform	Windows
Header	d3d10effect.h
Library	d3d10_1.lib, d3d10.lib
DLL	d3d10_1.dll, d3d10.dll

See also

Effect Functions (Direct3D 10)

D3D10CreateStateBlock function (d3d10effect.h)

Article • 10/13/20212 minutes to read

Create a state block.

Syntax

```
C++

HRESULT D3D10CreateStateBlock(
  [in]  ID3D10Device           *pDevice,
  [in]  D3D10_STATE_BLOCK_MASK *pStateBlockMask,
  [out] ID3D10StateBlock       **ppStateBlock
);
```

Parameters

`[in] pDevice`

Type: **ID3D10Device***

The device for which the state block will be created.

`[in] pStateBlockMask`

Type: **D3D10_STATE_BLOCK_MASK***

Indicates which parts of the device state will be captured when calling ID3D10StateBlock::Capture and reapplied when calling ID3D10StateBlock::Apply. See remarks.

`[out] ppStateBlock`

Type: **ID3D10StateBlock****

Address of a pointer to the buffer created (see ID3D10StateBlock Interface).

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

A state block is a collection of device state, and is used for saving and restoring device state. Use a state-block mask to enable subsets of state for saving and restoring.

The D3D10_STATE_BLOCK_MASK structure can be filled manually or by using any of the D3D10StateBlockMaskXXX APIs. A state block mask can also be obtained by calling ID3D10EffectTechnique::ComputeStateBlockMask or ID3D10EffectPass::ComputeStateBlockMask.

> Differences between Direct3D 9 and Direct3D 10:
> In Direct3D 10, a state block object does not contain any valid information about the state of the device until ID3D10StateBlock::Capture is called. In Direct3D 9, state is saved in a state block object, when it is created.

Requirements

Target Platform	Windows
Header	d3d10effect.h
Library	D3D10.lib
DLL	D3D10.dll

See also

Core Functions

Effect Functions

D3D10DisassembleEffect function (d3d10effect.h)

Article • 10/13/20212 minutes to read

This function -- which disassembles a compiled effect into a text string that contains assembly instructions and register assignments -- has been deprecated. Instead, use D3DDisassemble10Effect.

Syntax

```
C++

HRESULT D3D10DisassembleEffect(
  [in]  ID3D10Effect *pEffect,
  [in]  BOOL         EnableColorCode,
  [out] ID3D10Blob   **ppDisassembly
);
```

Parameters

`[in] pEffect`

Type: **ID3D10Effect***

A pointer to an ID3D10Effect Interface, which contains the compiled effect.

`[in] EnableColorCode`

Type: **BOOL**

Include HTML tags in the output to color code the result.

`[out] ppDisassembly`

Type: **ID3D10Blob****

A pointer to an ID3D10Blob Interface which contains the disassembled shader.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Remarks

This returned text includes a header with the version of the HLSL compiler used to generate this object, comments describing the memory layout of the constant buffers used by the shader, input and output signatures, and resource binding points.

Here is an example of disassembling a compiled effect. The example assumes you start with a compiled effect (shown as *l_pBlob_Effect* which you can see in Compile an Effect (Direct3D 10)).

```
LPCSTR commentString = NULL;
ID3D10Blob* pIDisassembly = NULL;
char* pDisassembly = NULL;
if( pVSBuf )
{
    D3D10DisassembleEffect( (UINT*) l_pBlob_Effect->GetBufferPointer(),
        l_pBlob_Effect->GetBufferSize(), TRUE, commentString, &pIDisassembly
);
    if( pIDisassembly )
    {
        FILE* pFile = fopen( "effect.htm", "w" );
        if( pFile)
        {
            fputs( (char*)pIDisassembly->GetBufferPointer(), pFile );
            fclose( pFile );
        }
    }
}
```

Requirements

Target Platform	Windows
Header	d3d10effect.h
Library	D3D10.lib
DLL	D3D10.dll

See also

Effect Functions (Direct3D 10)

D3D10StateBlockMaskDifference function (d3d10effect.h)

Article • 10/13/20212 minutes to read

Combine two state-block masks with a bitwise XOR.

Syntax

C++

```
HRESULT D3D10StateBlockMaskDifference(
  [in]  D3D10_STATE_BLOCK_MASK *pA,
  [in]  D3D10_STATE_BLOCK_MASK *pB,
  [out] D3D10_STATE_BLOCK_MASK *pResult
);
```

Parameters

`[in] pA`

Type: **D3D10_STATE_BLOCK_MASK***

State block mask on the left side of the bitwise XOR operation. See D3D10_STATE_BLOCK_MASK.

`[in] pB`

Type: **D3D10_STATE_BLOCK_MASK***

State block mask on the right side of the bitwise XOR operation.

`[out] pResult`

Type: **D3D10_STATE_BLOCK_MASK***

The result of the bitwise XOR operation.

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h
Library	D3D10.lib
DLL	D3D10.dll

See also

Core Functions

Effect Functions

D3D10StateBlockMaskDisableAll function (d3d10effect.h)

Article10/13/2021

Disable all state capturing with a state-block mask.

Syntax

```
C++

HRESULT D3D10StateBlockMaskDisableAll(
  [out] D3D10_STATE_BLOCK_MASK *pMask
);
```

Parameters

`[out] pMask`

Type: **D3D10_STATE_BLOCK_MASK***

A mask filled with all zeroes (see D3D10_STATE_BLOCK_MASK).

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h
Library	D3D10.lib
DLL	D3D10.dll

See also

Core Functions

Effect Functions

D3D10StateBlockMaskDisableCapture function (d3d10effect.h)

Article • 10/13/20212 minutes to read

Disable state capturing with a state-block mask.

Syntax

C++

```
HRESULT D3D10StateBlockMaskDisableCapture(
  [in, out] D3D10_STATE_BLOCK_MASK   *pMask,
  [in]      D3D10_DEVICE_STATE_TYPES StateType,
  [in]      UINT                     RangeStart,
  [in]      UINT                     RangeLength
);
```

Parameters

`[in, out] pMask`

Type: **D3D10_STATE_BLOCK_MASK***

A state block mask (see D3D10_STATE_BLOCK_MASK).

`[in] StateType`

Type: **D3D10_DEVICE_STATE_TYPES**

The type of device state to disable (see D3D10_DEVICE_STATE_TYPES).

`[in] RangeStart`

Type: **UINT**

The lower end of the range of values to set to false.

`[in] RangeLength`

Type: **UINT**

The upper end of the range of values to set to false.

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

This is an example of how to call this function. It creates a mask that cannot capture and apply to geometry-shader samplers in slots 2 ~ 23.

```
D3D10_STATE_BLOCK_MASK stateBlockMask;
D3D10StateBlockMaskDisableCapture(&stateBlockMask,
                                  D3D10_DST_GS_SAMPLERS,
                                  2, 23);
```

Requirements

Target Platform	Windows
Header	d3d10effect.h
Library	D3D10.lib
DLL	D3D10.dll

See also

Core Functions

Effect Functions

D3D10StateBlockMaskEnableAll function (d3d10effect.h)

Article • 10/13/20212 minutes to read

Enable a state-block mask to capture and apply all state variables.

Syntax

```
C++

HRESULT D3D10StateBlockMaskEnableAll(
  [out] D3D10_STATE_BLOCK_MASK *pMask
);
```

Parameters

`[out] pMask`

Type: **D3D10_STATE_BLOCK_MASK***

A mask with everything turned on.

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h
Library	D3D10.lib
DLL	D3D10.dll

See also

Core Functions

Effect Functions

D3D10StateBlockMaskEnableCapture function (d3d10effect.h)

Article10/13/2021

Enable a range of state values in a state block mask.

Syntax

C++

```
HRESULT D3D10StateBlockMaskEnableCapture(
  [in, out] D3D10_STATE_BLOCK_MASK   *pMask,
  [in]      D3D10_DEVICE_STATE_TYPES StateType,
  [in]      UINT                     RangeStart,
  [in]      UINT                     RangeLength
);
```

Parameters

`[in, out] pMask`

Type: **D3D10_STATE_BLOCK_MASK***

A state block mask (see D3D10_STATE_BLOCK_MASK).

`[in] StateType`

Type: **D3D10_DEVICE_STATE_TYPES**

The type of device state to enable (see D3D10_DEVICE_STATE_TYPES.

`[in] RangeStart`

Type: **UINT**

The lower end of the range of values to set to true.

`[in] RangeLength`

Type: **UINT**

The upper end of the range of values to set to true.

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

This is an example of how to call this function. It create a mask that can capture and apply to geometry-shader samplers in slots 2 ~ 13.

```
D3D10_STATE_BLOCK_MASK stateBlockMask;
D3D10StateBlockMaskEnableCapture(&stateBlockMask,
                                 D3D10_DST_GS_SAMPLERS,
                                 2, 13);
```

Requirements

Target Platform	Windows
Header	d3d10effect.h
Library	D3D10.lib
DLL	D3D10.dll

See also

Core Functions

Effect Functions

D3D10StateBlockMaskGetSetting function (d3d10effect.h)

Article • 10/13/20212 minutes to read

Get an element in a state-block mask; determine if an element is allowed by the mask for capturing and applying.

Syntax

```
C++

BOOL D3D10StateBlockMaskGetSetting(
  [in] D3D10_STATE_BLOCK_MASK   *pMask,
  [in] D3D10_DEVICE_STATE_TYPES StateType,
  [in] UINT                     Entry
);
```

Parameters

`[in] pMask`

Type: **D3D10_STATE_BLOCK_MASK***

State block mask. See D3D10_STATE_BLOCK_MASK.

`[in] StateType`

Type: **D3D10_DEVICE_STATE_TYPES**

The type of device state. See D3D10_DEVICE_STATE_TYPES.

`[in] Entry`

Type: **UINT**

The entry within the device state. This is only relevant for state types that have more than one entry, such as D3D10_DST_GS_SAMPLERS.

Return value

Type: **BOOL**

Returns true if the specified feature is allowed by the mask for capturing and applying, and false otherwise.

Requirements

Target Platform	Windows
Header	d3d10effect.h
Library	D3D10.lib
DLL	D3D10.dll

See also

Core Functions

Effect Functions

D3D10StateBlockMaskIntersect function (d3d10effect.h)

Article10/13/2021

Combine two state-block masks with a bitwise AND.

Syntax

C++

```
HRESULT D3D10StateBlockMaskIntersect(
  [in]  D3D10_STATE_BLOCK_MASK *pA,
  [in]  D3D10_STATE_BLOCK_MASK *pB,
  [out] D3D10_STATE_BLOCK_MASK *pResult
);
```

Parameters

`[in] pA`

Type: **D3D10_STATE_BLOCK_MASK***

State block mask on the left side of the bitwise AND operation. See D3D10_STATE_BLOCK_MASK.

`[in] pB`

Type: **D3D10_STATE_BLOCK_MASK***

State block mask on the right side of the bitwise AND operation.

`[out] pResult`

Type: **D3D10_STATE_BLOCK_MASK***

The result of the bitwise AND operation.

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h
Library	D3D10.lib
DLL	D3D10.dll

See also

Core Functions

Effect Functions

D3D10StateBlockMaskUnion function (d3d10effect.h)

Article10/13/2021

Combine two state-block masks with a bitwise OR.

Syntax

C++

```
HRESULT D3D10StateBlockMaskUnion(
  [in]  D3D10_STATE_BLOCK_MASK *pA,
  [in]  D3D10_STATE_BLOCK_MASK *pB,
  [out] D3D10_STATE_BLOCK_MASK *pResult
);
```

Parameters

`[in] pA`

Type: **D3D10_STATE_BLOCK_MASK***

State block mask on the left side of the bitwise OR operation. See D3D10_STATE_BLOCK_MASK.

`[in] pB`

Type: **D3D10_STATE_BLOCK_MASK***

State block mask on the right side of the bitwise OR operation.

`[out] pResult`

Type: **D3D10_STATE_BLOCK_MASK***

The result of the bitwise OR operation.

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Requirements

Target Platform	Windows
Header	d3d10effect.h
Library	D3D10.lib
DLL	D3D10.dll

See also

Core Functions

Effect Functions

Effect Structures (Direct3D 10)

Article • 01/06/2021 • 2 minutes to read

This section contains information about the following effect structures:

Structures	Description
D3D10_EFFECT_DESC	Describes an effect.
D3D10_EFFECT_SHADER_DESC	Describes an effect shader.
D3D10_EFFECT_TYPE_DESC	Describes an effect-variable type.
D3D10_EFFECT_VARIABLE_DESC	Describes an effect variable.
D3D10_PASS_DESC	Describes a pass.
D3D10_PASS_SHADER_DESC	Describes a shader variable.
D3D10_TECHNIQUE_DESC	Describes a technique.

Related topics

Effect Reference

D3D10_EFFECT_DESC structure (d3d10effect.h)

Article • 04/02/20212 minutes to read

Describes an effect.

Syntax

C++

```
typedef struct _D3D10_EFFECT_DESC {
  BOOL IsChildEffect;
  UINT ConstantBuffers;
  UINT SharedConstantBuffers;
  UINT GlobalVariables;
  UINT SharedGlobalVariables;
  UINT Techniques;
} D3D10_EFFECT_DESC;
```

Members

`IsChildEffect`

Type: **BOOL**

TRUE if the effect is a child effect; otherwise **FALSE**.

`ConstantBuffers`

Type: **UINT**

The number of constant buffers.

`SharedConstantBuffers`

Type: **UINT**

The number of constant buffers shared in an effect pool.

`GlobalVariables`

Type: **UINT**

The number of global variables.

`SharedGlobalVariables`

Type: **UINT**

The number of global variables shared in an effect pool.

`Techniques`

Type: **UINT**

The number of techniques.

Remarks

To get an effect description, call ID3D10Effect::GetDesc.

Requirements

Header	d3d10effect.h (include D3D10.h)

See also

Effect Structures (Direct3D 10)

D3D10_EFFECT_SHADER_DESC structure (d3d10effect.h)

Article • 08/03/20212 minutes to read

Describes an effect shader.

Syntax

C++

```
typedef struct _D3D10_EFFECT_SHADER_DESC {
  const BYTE *pInputSignature;
  BOOL       IsInline;
  const BYTE *pBytecode;
  UINT       BytecodeLength;
  LPCSTR     SODecl;
  UINT       NumInputSignatureEntries;
  UINT       NumOutputSignatureEntries;
} D3D10_EFFECT_SHADER_DESC;
```

Members

`pInputSignature`

Type: **const BYTE***

Passed into CreateInputLayout. Only valid on a vertex shader or geometry shader. See ID3D10Device_CreateInputLayout.

`IsInline`

Type: **BOOL**

TRUE is the shader is defined inline; otherwise **FALSE**.

`pBytecode`

Type: **const BYTE***

A pointer to the compiled shader.

`BytecodeLength`

Type: **UINT**

The length of pBytecode.

`SODecl`

Type: **LPCSTR**

A string that contains a declaration of the stream output from a geometry shader.

`NumInputSignatureEntries`

Type: **UINT**

The number of entries in the input signature.

`NumOutputSignatureEntries`

Type: **UINT**

The number of entries in the output signature.

Remarks

To get an effect-shader description, call ID3D10EffectShaderVariable::GetShaderDesc.

Requirements

Header	d3d10effect.h (include D3D10.h)

See also

Effect Structures (Direct3D 10)

D3D10_EFFECT_TYPE_DESC structure (d3d10effect.h)

Article • 04/02/20212 minutes to read

Describes an effect-variable type.

Syntax

C++

```
typedef struct _D3D10_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;
} D3D10_EFFECT_TYPE_DESC;
```

Members

`TypeName`

Type: **LPCSTR**

A string that contains the variable name.

`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**

The number of elements if the variable is an array; otherwise 0.

`Members`

Type: **UINT**

The number of members if the variable is a structure; otherwise 0.

`Rows`

Type: **UINT**

The number of rows if the variable is a matrix; otherwise 0.

`Columns`

Type: **UINT**

The number of columns if the variable is a matrix; otherwise 0.

`PackedSize`

Type: **UINT**

The number of bytes that the variable consumes when it is packed tightly by the compiler.

`UnpackedSize`

Type: **UINT**

The number of bytes that the variable consumes before it is packed by the compiler.

`Stride`

Type: **UINT**

The number of bytes between elements.

Remarks

To get an effect-variable type, call ID3D10EffectType::GetDesc.

Requirements

Header	d3d10effect.h (include D3D10.h)

See also

Effect Structures (Direct3D 10)

D3D10_EFFECT_VARIABLE_DESC structure (d3d10effect.h)

Article04/02/2021

Describes an effect variable.

Syntax

C++

```
typedef struct _D3D10_EFFECT_VARIABLE_DESC {
  LPCSTR Name;
  LPCSTR Semantic;
  UINT   Flags;
  UINT   Annotations;
  UINT   BufferOffset;
  UINT   ExplicitBindPoint;
} D3D10_EFFECT_VARIABLE_DESC;
```

Members

`Name`

Type: **LPCSTR**

A string that contains the variable name.

`Semantic`

Type: **LPCSTR**

The semantic attached to the variable; otherwise **NULL**.

`Flags`

Type: **UINT**

Optional flags for effect variables.

`Annotations`

Type: **UINT**

The number of annotations; otherwise 0.

`BufferOffset`

Type: **UINT**

The offset between the beginning of the constant buffer and this variable; otherwise 0.

`ExplicitBindPoint`

Type: **UINT**

The register that this variable is bound to. To bind a variable explicitly use the D3D10_EFFECT_VARIABLE_EXPLICIT_BIND_POINT flag.

Remarks

To get an effect-variable description, call ID3D10EffectVariable::GetDesc.

Requirements

Header	d3d10effect.h (include D3D10.h)

See also

Effect Structures (Direct3D 10)

D3D10_PASS_DESC structure (d3d10effect.h)

Article • 09/01/20222 minutes to read

Describes an effect pass, which contains pipeline state.

Syntax

C++

```
typedef struct _D3D10_PASS_DESC {
  LPCSTR Name;
  UINT   Annotations;
  BYTE   *pIAInputSignature;
  SIZE_T IAInputSignatureSize;
  UINT   StencilRef;
  UINT   SampleMask;
  FLOAT  BlendFactor[4];
} D3D10_PASS_DESC;
```

Members

`Name`

Type: **LPCSTR**

A string that contains the name of the pass; otherwise **NULL**.

`Annotations`

Type: **UINT**

The number of annotations.

`pIAInputSignature`

Type: **BYTE***

A pointer to the input signature or the vertex shader; otherwise **NULL**.

`IAInputSignatureSize`

Type: **SIZE_T**

The size of the input signature (in bytes).

`StencilRef`

Type: **UINT**

The stencil-reference value used in the depth-stencil state (see Configuring Depth-Stencil Functionality (Direct3D 10)).

`SampleMask`

Type: **UINT**

The sample mask for the blend state (see Configuring Blending Functionality (Direct3D 10)).

`BlendFactor[4]`

Type: **FLOAT**

The per-component blend factors (RGBA) for the blend state (see Configuring Blending Functionality (Direct3D 10)).

Remarks

Get a pass description by calling ID3D10EffectPass::GetDesc; an effect technique contains one or more passes.

Requirements

Header	d3d10effect.h (include D3D10.h)

See also

Effect Structures (Direct3D 10)

D3D10_PASS_SHADER_DESC structure (d3d10effect.h)

Article • 04/02/20212 minutes to read

Describes an effect variable that contains a shader.

Syntax

C++

```
typedef struct _D3D10_PASS_SHADER_DESC {
  ID3D10EffectShaderVariable *pShaderVariable;
  UINT                       ShaderIndex;
} D3D10_PASS_SHADER_DESC;
```

Members

`pShaderVariable`

Type: **ID3D10EffectShaderVariable***

A pointer to the variable that the shader came from. If it 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, this will point to a shader variable that returns false when IsValid is called.

`ShaderIndex`

Type: **UINT**

A zero-based array index; otherwise 0.

Remarks

To get a shader description, call a method like ID3D10EffectPass::GetVertexShaderDesc.

Requirements

Header	d3d10effect.h (include D3D10.h)

See also

Effect Structures (Direct3D 10)

D3D10_TECHNIQUE_DESC structure (d3d10effect.h)

Article • 04/02/20212 minutes to read

Describes an effect technique.

Syntax

C++

```
typedef struct _D3D10_TECHNIQUE_DESC {
  LPCSTR Name;
  UINT   Passes;
  UINT   Annotations;
} D3D10_TECHNIQUE_DESC;
```

Members

`Name`

Type: **LPCSTR**

A string that contains the technique name; otherwise **NULL**.

`Passes`

Type: **UINT**

The number of passes in the technique.

`Annotations`

Type: **UINT**

The number of annotations.

Remarks

To get a technique, call ID3D10EffectTechnique::GetDesc.

Requirements

Header	d3d10effect.h (include D3D10.h)

See also

Effect Structures (Direct3D 10)

Effect Constants (Direct3D 10)

Article • 01/06/2021 • 2 minutes to read

This section contains information about the effect constants:

Constants	Description
D3D10_EFFECT Constants	These constants are effect compiler options.
D3D10_EFFECT_VARIABLE Constants	These constants are effect variable properties.

Related topics

Effect Reference

D3D10_EFFECT Constants

Article • 01/06/2021 • 2 minutes to read

These constants used when creating an effect to define either compilation behavior or runtime effect behavior.

#define	Value	Description
D3D10_EFFECT_COMPILE_CHILD_EFFECT	1 << 0	Compile the .fx file to a child effect. Child effects have no initializes for any shared values because these are initialized in the effect pool.
D3D10_EFFECT_COMPILE_ALLOW_SLOW_OPS	1 << 1	By default, performance mode is enabled. Performance mode disallows mutable state objects by preventing non-literal expressions from appearing in state object definitions. Specifying this flag will disable the mode and allow for mutable state objects.
D3D10_EFFECT_SINGLE_THREADED	1 << 3	Do not attempt to synchronize with other threads loading effects into the same pool.

These constants are defined as macros in d3d10effect.h.

Related topics

Effect Constants (Direct3D 10)

D3D10_EFFECT_VARIABLE Constants

Article • 01/06/2021 • 2 minutes to read

These constants apply to effect variables.

#define	Description	Value
D3D10_EFFECT_VARIABLE_ANNOTATION	1 << 0	Indicates that this is an annotation or a global variable.
D3D10_EFFECT_VARIABLE_POOLED	1 << 1	Indicates that this variable or constant buffer resides in an effect pool. If this flag is not set, then the variable resides in a standalone effect or a child effect (standalone effects return false from **ID3D10Effect::IsPool**.
D3D10_EFFECT_VARIABLE_EXPLICIT_BIND_POINT	1 << 2	Indicates that the variable has been explicitly bound using the register keyword.

These flags are defined as macros in d3d10effect.h.

Related topics

Effect Constants (Direct3D 10)

Effect Format (Direct3D 10)

Article • 01/06/2021 • 2 minutes to read

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.
- A technique, which implements 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.

Related topics

Effect Reference

Effect Variable Syntax (Direct3D 10)

Article • 01/06/2021 • 2 minutes to read

An effect variable is declared with the following syntax.

Syntax

DataType VariableName [: *SemanticName*] < *Annotations* >;

Name	Description
DataType	Any basic or texture type.
VariableName	An ASCII string that uniquely identifies the name of the effect variable.
SemanticName	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.
Annotations	One or more pieces of user-supplied information (metadata) that is ignored by the effect system. For syntax, see Annotation Syntax (Direct3D 10).

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

The BasicHLSL10 sample uses global variables without semantics for material colors, light properties and transformation matrices.

This example illustrates global effect 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 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.

Related topics

Effect Format

Annotation Syntax (Direct3D 10)

Article • 01/06/2021 • 2 minutes to read

An annotation is a user-defined piece of information, declared with the following syntax.

<DataType Name = Value; ... ;>

Parameters

Item	Description
DataType	[in] The data type, which includes any scalar HLSL type as well as the string type.
Name	[in] An ASCII string, that represents the annotation name.
Value	[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 Variable Syntax

Effect Sampler Syntax (Deprecated, Direct3D 10)

Article • 01/06/2021 • 2 minutes to read

This page has been combined to show the differences between Direct3D 9 and Direct3D 10 syntax, it has been moved to Sampler Type (DirectX HLSL).

Related topics

Effect Variable Syntax

Effect Function Syntax (Direct3D 10)

Article • 01/06/2021 • 2 minutes to read

An effect function is written in HLSL and is declared with the following syntax.

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 BasicHLSL10 sample uses both a pixel shader and a vertex shader. The pixel shader function is called RenderScenePS and is shown below.

```
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 10)

Article • 08/20/2021 • 2 minutes to read

An effect technique is declared with the following syntax.

technique10 *TechniqueName* [<*Annotations* >]

{
pass *PassName* \[<*Annotations* > \] {
\[*SetStateGroup*; \] \[*SetStateGroup*; \] ... \[*SetStateGroup*; \]
}

}

Parameters

technique10

Required keyword.

TechniqueName

Optional. An ASCII string that uniquely identifies the name of the effect technique.

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 10).

pass

Required keyword.

PassName

[in] Optional. An ASCII string that uniquely identifies the name of the pass.

SetStateGroup

[in] Set one or more state groups such as:

StateGroup	Syntax

StateGroup	Syntax
Blend State	`SetBlendState( arguments );` See [**OMSetBlendState**](/windows/desktop/api/D3D10/nf-d3d10-id3d10device-omsetblendstate) for the argument list.
Depth-stencil State	`SetDepthStencilState( arguments );` See **OMSetDepthStencilState** for the argument list.
Rasterizer State	`SetRasterizerState( arguments );` See [**RSSetState**](/windows/desktop/api/D3D10/nf-d3d10-id3d10device-rssetstate) for the argument list.

StateGroup	Syntax

Shader State	`SetXXXShader( CompileShader( shader_profile, ShaderFunction( args ) ) );`
	or
	`SetXXXShader( CompileShader( NULL ) );`
	or
	`SetXXXShader( NULL );`
	SetXXXShader is one of **SetVertexShader**, **SetGeometryShader**, or **SetPixelShader** (which are similar to the API methods **VSSetShader**, **GSSetShader**, and **PSSetShader**).

Effect state groups are listed in effect state.

Examples

This example (from CubeMapGS sample) 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 (from BasicHLSL10 sample); which uses a vertex and pixel shader.

```
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 state groups (Direct3D 10)

Article • 05/24/2021 • 2 minutes to read

Effect states are name-value pairs in the form of an expression.

Blend state

Effect state	Group
ALPHATOCOVERAGEENABLE, BLENDENABLE, SRCBLEND, DESTBLEND, BLENDOP, SRCBLENDALPHA, DESTBLENDALPHA, BLENDOPALPHA, RENDERTARGETWRITEMASK	Members of D3D10_BLEND_DESC

Depth and stencil state

Effect state	Group
DEPTHENABLE, DEPTHWRITEMASK, DEPTHFUNC, STENCILENABLE, STENCILREADMASK, STENCILWRITEMASK	Members of D3D10_DEPTH_STENCIL_DESC
FRONTFACESTENCILFAIL, FRONTFACESTENCILZFAIL, FRONTFACESTENCILPASS, FRONTFACESTENCILFUNC, BACKFACESTENCILFAIL, BACKFACESTENCILZFAIL, BACKFACESTENCILPASS, BACKFACESTENCILFUNC	Member of D3D10_DEPTH_STENCILOP_DESC

Rasterizer state

Effect state	Group
FILLMODE	D3D10_FILL_MODE
CULLMODE	D3D10_CULL_MODE
FRONTCOUNTERCLOCKWISE, DEPTHBIAS, DEPTHBIASCLAMP, SLOPESCALEDDEPTHBIAS, ZCLIPENABLE, SCISSORENABLE, MULTISAMPLEENABLE, ANTIALIASEDLINEENABLE	Members of D3D10_RASTERIZER_DESC

Sampler state

Effect state	Group

Effect state	Group
Filter, **AddressU**, **AddressV**, **AddressW**, **MipLODBias**, **MaxAnisotropy**, **ComparisonFunc**, **BorderColor**, **MinLOD**, **MaxLOD**	Members of **D3D10_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_STENCILREF AB_BLENDFACTOR AB_SAMPLEMASK	Members of **D3D10_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 ObjectName {
 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 10). 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 );
```

For a tutorial describing the use of states see State management.

Related topics

- Effect Technique Syntax
- Effect Format (Direct3D 10)

Direct3D 10 Return Codes

Article • 01/06/2021 • 2 minutes to read

The following table contains return codes from API functions.

HRESULT	Description
D3D10_ERROR_FILE_NOT_FOUND	The file was not found.
D3D10_ERROR_TOO_MANY_UNIQUE_STATE_OBJECTS	There are too many unique instances of a particular type of state object.
D3DERR_INVALIDCALL	The method call is invalid. For example, a method's parameter may not be a valid pointer.
D3DERR_WASSTILLDRAWING	The previous blit operation that is transferring information to or from this surface is incomplete.
E_FAIL	Attempted to create a device with the debug layer enabled and the layer is not installed.
E_INVALIDARG	An invalid parameter was passed to the returning function.
E_OUTOFMEMORY	Direct3D could not allocate sufficient memory to complete the call.
E_NOTIMPL	The method call isn't implemented with the passed parameter combination.
S_FALSE	Alternate success value, indicating a successful but nonstandard completion (the precise meaning depends on context).
S_OK	No error occurred.

To write code that handles HRESULT values robustly, use the SUCCEEDED(hr) and FAILED(hr) macros.

Related topics

Direct3D Reference

Reference for Direct3D 10

D3DX Reference (Direct3D 10 Graphics)

Article • 06/19/2021 • 2 minutes to read

> **Note**
>
> The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated for Windows 8, and is not supported for Microsoft Store apps. If upgrading to a newer version of Direct3D and associated utility code is not an option, you can make use of the **Microsoft.DXSDK.D3DX** NuGet package rather than relying on the legacy DirectX SDK or DirectSetup.

This section contains the following sections:

- D3DX Interfaces
- D3DX Functions
- D3DX Structures
- D3DX Enumerations

Related topics

- Reference for Direct3D 10

D3DX Interfaces (Direct3D 10 Graphics)

Article • 06/19/2021 • 2 minutes to read

This section contains reference information for the component object model (COM) interfaces provided by the D3DX utility library. The following interfaces are used with the D3DX utility library.

Interfaces	Description
ID3DX10DataLoader Interface	Data loading object used by ID3DX10ThreadPump Interface for loading data asynchronously.
ID3DX10DataProcessor Interface	Data processing object used by ID3DX10ThreadPump Interface for processing loaded data asynchronously.
ID3DX10Font Interface	The ID3DX10Font interface encapsulates the textures and resources needed to render a specific font on a specific device.
ID3DX10Mesh Interface	Applications use the methods of the ID3DX10Mesh interface to manipulate mesh objects.
ID3DX10MeshBuffer Interface	
ID3DX10SkinInfo Interface	ID3DX10SkinInfo allows you to optimize, process, and manually set the relationship between bones and vertices in your meshes (see Skeletal Animation on Wikipedia). It is most useful for making .x files exported by DCC Apps (such as 3DS Max and Maya) more hardware-friendly, and for improving the render speed of your skinned meshes in software render mode.
ID3DX10Sprite Interface	The ID3DX10Sprite interface provides a set of methods that simplify the process of drawing sprites using Microsoft Direct3D.
ID3DX10ThreadPump Interface	Used to execute tasks asynchronously. This object takes up a substantial amount of resources, so generally only one should be created per application.
ID3DXMatrixStack Interface	Applications use the methods of the ID3DXMATRIXStack interface to manipulate a matrix stack.

Related topics

D3DX Reference

ID3DX10DataLoader interface

Article • 03/15/2021 • 2 minutes to read

Data loading object used by **ID3DX10ThreadPump Interface** for loading data asynchronously.

Members

The **ID3DX10DataLoader** interface inherits from the **IUnknown** interface. **ID3DX10DataLoader** also has these types of members:

- Methods

Methods

The **ID3DX10DataLoader** interface has these methods.

Method	Description
Decompress	Used to decompress encoded data. Typically this would be used 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.
Destroy	Used by a **ID3DX10ThreadPump Interface** to destroy the loader after a work item completes.
Load	Used by a **ID3DX10ThreadPump Interface** to load data from a disk.

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
Header	D3DX10.h
Library	D3DX10.lib

See also

D3DX Interfaces

ID3DX10DataLoader::Decompress method

Article • 03/15/2021 • 2 minutes to read

Used to decompress encoded data. Typically this would be used 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.

Syntax

```
C++

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 10 Return Codes.

Remarks

ID3DX10DataLoader Interface can be inherited and its members redefined. Decompress could be redefined to support your own custom file formats.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10DataLoader

D3DX Interfaces

ID3DX10DataLoader::Destroy method

Article • 03/15/2021 • 2 minutes to read

Used by a **ID3DX10ThreadPump Interface** to destroy the loader after a work item completes.

Syntax

```
C++

HRESULT Destroy();
```

Parameters

This method has no parameters.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10DataLoader

D3DX Interfaces

ID3DX10DataLoader::Load method

Article • 03/15/2021 • 2 minutes to read

Used by a **ID3DX10ThreadPump Interface** to load data from a disk.

Syntax

```
C++

HRESULT Load();
```

Parameters

This method has no parameters.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10DataLoader

D3DX Interfaces

ID3DX10DataProcessor interface

Article • 03/15/2021 • 2 minutes to read

Data processing object used by **ID3DX10ThreadPump Interface** for processing loaded data asynchronously.

Members

The **ID3DX10DataProcessor** interface inherits from the **IUnknown** interface. **ID3DX10DataProcessor** also has these types of members:

- Methods

Methods

The **ID3DX10DataProcessor** interface has these methods.

Method	Description
CreateDeviceObject	Create a device object.
Destroy	Used by a **ID3DX10ThreadPump Interface** to destroy the processor after a work item completes.
Process	Process some data.

Remarks

This object can be inherited and its members redefined. Doing so would enable you to customize the API for processing your own custom file formats.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

D3DX Interfaces

ID3DX10DataProcessor::CreateDeviceObject method

Article • 03/15/2021 • 2 minutes to read

Create a device object.

Syntax

```
C++

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 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10DataProcessor

ID3DX10DataProcessor::Destroy method

Article • 03/15/2021 • 2 minutes to read

Used by a **ID3DX10ThreadPump Interface** to destroy the processor after a work item completes.

Syntax

```
C++

HRESULT Destroy();
```

Parameters

This method has no parameters.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10DataProcessor

D3DX Interfaces

ID3DX10DataProcessor::Process method

Article • 03/15/2021 • 2 minutes to read

Process some data.

Syntax

```
C++

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 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10DataProcessor

D3DX Interfaces

ID3DX10Font interface

Article • 03/15/2021 • 2 minutes to read

The ID3DX10Font interface encapsulates the textures and resources needed to render a specific font on a specific device.

Members

The **ID3DX10Font** interface inherits from the IUnknown interface. **ID3DX10Font** also has these types of members:

- Methods

Methods

The **ID3DX10Font** interface has these methods.

Method	Description
DrawText	Draw formatted text. This method supports ANSI and Unicode strings.
GetDC	Return a handle to a display device context (DC) that has the font set onto it.
GetDesc	Get a description of the current font object.
GetDevice	Retrieve the Direct3D device associated with the font object.
GetGlyphData	Return information about the placement and orientation of a glyph in a character cell.
GetTextMetrics	Retrieve font characteristics.
PreloadCharacters	Load a series of characters into video memory to improve the efficiency of rendering to the device.
PreloadGlyphs	Load a series of glyphs into video memory to improve the efficiency of rendering to the device.
PreloadText	Load formatted text into video memory to improve the efficiency of rendering to the device. This method supports ANSI and Unicode strings.

Remarks

The ID3DX10Font interface is obtained by calling **D3DX10CreateFont** or **D3DX10CreateFontIndirect**.

The LPD3DX10FONT type is defined as a pointer to the ID3DX10Font interface.

```
typedef interface ID3DX10Font ID3DX10Font;
typedef interface ID3DX10Font *LPD3DX10FONT;
```

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

D3DX Interfaces

ID3DX10Font::DrawText method

Article • 03/15/2021 • 3 minutes to read

Draw formatted text. This method supports ANSI and Unicode strings.

Syntax

```
C++

INT DrawText(
  [in] LPD3DX10SPRITE pSprite,
  [in] LPCTSTR        pString,
  [in] INT            Count,
  [in] LPRECT         pRect,
  [in] UINT           Format,
  [in] D3DXCOLOR      Color
);
```

Parameters

pSprite [in]

Type: **LPD3DX10SPRITE**

Pointer to an ID3DX10Sprite object that contains the string you wish to draw. Can be **NULL**, in which case Direct3D will render the string with its own sprite object. To improve efficiency, a sprite object should be specified if ID3DX10Font::DrawText is to be called more than once in a row.

pString [in]

Type: **LPCTSTR**

Pointer to a string to draw. If UNICODE is defined, this parameter type resolves to an LPCWSTR, otherwise, the type resolves to an LPCSTR. If the Count parameter is -1, the string must be **NULL** terminated.

Count [in]

Type: **INT**

The number of characters in the string. If Count is -1, then the pString parameter is assumed to be a pointer to a sprite containing a NULL-terminated string and

ID3DX10Font::DrawText computes the character count automatically.

pRect [in]

Type: **LPRECT**

Pointer to a RECT structure that contains the rectangle, in logical coordinates, in which the text is to be formatted. As with any RECT object, the coordinate value of the rectangle's right side must be greater than that of its left side. Likewise, the coordinate value of the bottom must be greater than that of the top.

Format [in]

Type: **UINT**

Specify the method of formatting the text. It can be any combination of the following values:

Item	Description
DT_BOTTOM	Justify the text to the bottom of the rectangle. This value must be combined with DT_SINGLELINE.
DT_CALCRECT	Tell DrawText to automatically calculate the width and height of the rectangle based on the length of the string you tell it to draw. If there are multiple lines of text, ID3DX10Font::DrawText uses the width of the rectangle pointed to by the pRect parameter and extends the base of the rectangle to bound the last line of text. If there is only one line of text, ID3DX10Font::DrawText modifies the right side of the rectangle so that it bounds the last character in the line. In either case, ID3DX10Font::DrawText returns the height of the formatted text but does not draw the text.
DT_CENTER	Center text horizontally in the rectangle.
DT_EXPANDTABS	Expand tab characters. The default number of characters per tab is eight.
DT_LEFT	Align text to the left.
DT_NOCLIP	Draw without clipping. ID3DX10Font::DrawText is somewhat faster when DT_NOCLIP is used.
DT_RIGHT	Align text to the right.
DT_RTLREADING	Display text in right-to-left reading order for bidirectional text when a Hebrew or Arabic font is selected. The default reading order for all text is left-to-right.
DT_SINGLELINE	Display text on a single line only. Carriage returns and line feeds do not break the line.
DT_TOP	Top-justify text.

Item	Description
DT_VCENTER	Center text vertically (single line only).
DT_WORDBREAK	Break words. Lines are automatically broken between words if a word would extend past the edge of the rectangle specified by the pRect parameter. A carriage return/line feed sequence also breaks the line.

Color [in]

Type: **D3DXCOLOR**

Color of the text. See **D3DXCOLOR**.

Return value

Type: **INT**

If the function succeeds, the return value is the height of the text in logical units. If DT_VCENTER or DT_BOTTOM is specified, the return value is the offset from pRect (top to the bottom) of the drawn text. If the function fails, the return value is zero.

Remarks

The parameters of this method are very similar to those of the GDI DrawText function.

This method supports both ANSI and Unicode strings.

Unless the DT_NOCLIP format is used, this method clips the text so that it does not appear outside the specified rectangle. All formatting is assumed to have multiple lines unless the DT_SINGLELINE format is specified.

If the selected font is too large for the rectangle, this method does not attempt to substitute a smaller font.

This method supports only fonts whose escapement and orientation are both zero.

Requirements

Requirement	Value
Header	D3DX10.h

Requirement	Value
Library	D3DX10.lib

See also

ID3DX10Font

D3DX Interfaces

ID3DX10Font::GetDC method

Article • 03/15/2021 • 2 minutes to read

Return a handle to a display device context (DC) that has the font set onto it.

Syntax

```
C++

HDC GetDC();
```

Parameters

This method has no parameters.

Return value

Type: **HDC**

Handle to a display DC.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Font

D3DX Interfaces

ID3DX10Font::GetDesc method

Article • 03/15/2021 • 2 minutes to read

Get a description of the current font object.

Syntax

```
C++

HRESULT GetDesc(
  [in] D3DX10_FONT_DESC *pDesc
);
```

Parameters

pDesc [in]

Type: **D3DX10_FONT_DESC***

Pointer to a **D3DX10_FONT_DESC** structure that describes the font object. If UNICODE is defined, a pointer to a D3DX10FONT_DESCW is returned; otherwise a pointer to a D3DX10FONT_DESCA is returned.

Return value

Type: **HRESULT**

If the method succeeds, the return value is S_OK. If the method fails, the following value will be returned: D3DERR_INVALIDCALL.

Remarks

This method describes Unicode font objects if UNICODE is defined. Otherwise GetDescA is called, which returns a pointer to the D3DX10FONT_DESCA structure.

Requirements

Requirement	Value

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Font

D3DX Interfaces

ID3DX10Font::GetDevice method

Article • 03/15/2021 • 2 minutes to read

Retrieve the Direct3D device associated with the font object.

Syntax

```
C++

HRESULT GetDevice(
  [out] ID3D10Device **ppDevice
);
```

Parameters

ppDevice [out]

Type: **ID3D10Device****

Address of a pointer to an ID3D10Device interface, representing the Direct3D device object associated with the font object.

Return value

Type: **HRESULT**

If the method succeeds, the return value is S_OK. If the method fails, the return value can be one of the following: D3DERR_INVALIDCALL, D3DXERR_INVALIDDATA.

Remarks

> **Note**
>
> Calling this method will increase the internal reference count on the ID3D10Device interface. Be sure to call IUnknown when you are done using this ID3D10Device interface or you will have a memory leak.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Font

D3DX Interfaces

ID3DX10Font::GetGlyphData method

Article • 11/19/2022 • 2 minutes to read

Return information about the placement and orientation of a glyph in a character cell.

Syntax

```
C++

HRESULT GetGlyphData(
  [in]  UINT                     Glyph,
  [out] ID3D10ShaderResourceView **ppTexture,
  [in]  RECT                     *pBlackBox,
  [in]  POINT                    *pCellInc
);
```

Parameters

Glyph [in]

Type: **UINT**

Glyph identifier.

ppTexture [out]

Type: **ID3D10ShaderResourceView****

Address of a pointer to a ID3D10Texture object that contains the glyph.

pBlackBox [in]

Type: **RECT***

Pointer to the smallest rectangle object that completely encloses the glyph. See RECT.

pCellInc [in]

Type: **POINT***

Pointer to the two-dimensional vector that connects the origin of the current character cell to the origin of the next character cell. See POINT.

Return value

Type: **HRESULT**

If the method succeeds, the return value is S_OK. If the method fails, the return value can be one of the following: D3DERR_INVALIDCALL, D3DXERR_INVALIDDATA.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Font

D3DX Interfaces

ID3DX10Font::GetTextMetrics method

Article • 03/15/2021 • 2 minutes to read

Retrieve font characteristics.

Syntax

```
C++

BOOL GetTextMetrics(
  [in] TEXTMETRIC *pTextMetrics
);
```

Parameters

pTextMetrics [in]

Type: **TEXTMETRIC***

Pointer to a TEXTMETRIC structure, which contains font properties. If Unicode is defined, the function returns a TEXTMETRICW structure. Otherwise, the function returns a TEXTMETRICA structure.

Return value

Type: **BOOL**

Nonzero if the function is successful; otherwise 0.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Font

ID3DX10Font::PreloadCharacters method

Article • 03/15/2021 • 2 minutes to read

Load a series of characters into video memory to improve the efficiency of rendering to the device.

Syntax

```C++
HRESULT PreloadCharacters(
  [in] UINT First,
  [in] UINT Last
);
```

Parameters

First [in]

Type: **UINT**

ID of the first character to be loaded into video memory.

Last [in]

Type: **UINT**

ID of the last character to be loaded into video memory.

Return value

Type: **HRESULT**

If the method succeeds, the return value is S_OK. If the method fails, the return value can be one of the following: D3DERR_INVALIDCALL, D3DXERR_INVALIDDATA.

Remarks

This method generates textures containing glyphs that represent the input characters. The glyphs are drawn as a series of triangles.

Characters will not be rendered to the device; ID3DX10Font::DrawText must still be called to render the characters. However, by pre-loading characters into video memory, ID3DX10Font::DrawText will use substantially fewer CPU resources.

This method internally converts characters to glyphs using the GDI function GetCharacterPlacement.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Font

D3DX Interfaces

ID3DX10Font::PreloadGlyphs method

Article • 03/15/2021 • 2 minutes to read

Load a series of glyphs into video memory to improve the efficiency of rendering to the device.

Syntax

```
C++

HRESULT PreloadGlyphs(
  [in] UINT First,
  [in] UINT Last
);
```

Parameters

First [in]

Type: **UINT**

ID of the first glyph to be loaded into video memory.

Last [in]

Type: **UINT**

ID of the last glyph to be loaded into video memory.

Return value

Type: **HRESULT**

If the method succeeds, the return value is S_OK. If the method fails, the return value can be one of the following: D3DERR_INVALIDCALL, D3DXERR_INVALIDDATA.

Remarks

This method generates textures that contain the input glyphs. The glyphs are drawn as a series of triangles.

Glyphs will not be rendered to the device; ID3DX10Font::DrawText must still be called to render the glyphs. However, by pre-loading glyphs into video memory, ID3DX10Font::DrawText will use substantially fewer CPU resources.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Font

D3DX Interfaces

ID3DX10Font::PreloadText method

Article • 03/15/2021 • 2 minutes to read

Load formatted text into video memory to improve the efficiency of rendering to the device. This method supports ANSI and Unicode strings.

Syntax

```C++
HRESULT PreloadText(
  [in] LPCSTR pString,
  [in] INT    Count
);
```

Parameters

pString [in]

Type: **LPCSTR**

Pointer to a string of characters to be loaded into video memory. If the compiler settings require Unicode, the data type LPCTSTR resolves to LPCWSTR; otherwise, the data type resolves to LPCSTR. See Remarks.

Count [in]

Type: **INT**

Number of characters in the text string.

Return value

Type: **HRESULT**

If the method succeeds, the return value is S_OK. If the method fails, the return value can be one of the following: D3DERR_INVALIDCALL, D3DXERR_INVALIDDATA.

Remarks

The compiler setting also determines the function version. If Unicode is defined, the function call resolves to PreloadTextW. Otherwise, the function call resolves to PreloadTextA because ANSI strings are being used.

This method generates textures that contain glyphs that represent the input text. The glyphs are drawn as a series of triangles.

Text will not be rendered to the device; ID3DX10Font::DrawText must still be called to render the text. However, by preloading text into video memory, ID3DX10Font::DrawText will use substantially fewer CPU resources.

This method internally converts characters to glyphs using the GDI function GetCharacterPlacement.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Font

D3DX Interfaces

ID3DX10Mesh interface

Article • 03/15/2021 • 2 minutes to read

Applications use the methods of the ID3DX10Mesh interface to manipulate mesh objects.

Members

The **ID3DX10Mesh** interface inherits from the IUnknown interface. **ID3DX10Mesh** also has these types of members:

- Methods

Methods

The **ID3DX10Mesh** interface has these methods.

Method	Description
CloneMesh	Creates a new mesh and fills it with the data of a previously loaded mesh.
CommitToDevice	Commit any changes made to a mesh to the device so that the changes can be rendered. This should be called after a mesh's data is altered and before it is rendered. A mesh cannot be rendered unless it is committed to the device. See remarks.
Discard	Removes mesh data from the device that has been committed to the device (with ID3DX10Mesh::CommitToDevice).
DrawSubset	Draws a subset of a mesh.
DrawSubsetInstanced	Draw several instances of the same subset of a mesh.
GenerateAdjacencyAndPointReps	Generate a list of mesh edges, as well as a list of faces that share each edge.
GenerateAttributeBufferFromTable	Generate an attribute buffer from the data in the mesh's attribute table. An attribute buffer is another format for storing the data in the attribute table. Both the attribute buffer and the attribute table are internal data structures in the mesh.

Method	Description
GenerateGSAdjacency	Adds adjacency data to the mesh's index buffer. When the mesh is to be sent to a geometry shader that takes adjacency data, it is neccessary for the mesh's index buffer to contain adjacency data.
GetAdjacencyBuffer	Access the mesh's adjacency buffer.
GetAttributeBuffer	Access the mesh's attribute buffer.
GetAttributeTable	Retrieves either an attribute table for a mesh, or the number of entries stored in an attribute table for a mesh.
GetDeviceIndexBuffer	Access the mesh's index buffer after it has been committed to the device with ID3DX10Mesh::CommitToDevice. This is different from ID3DX10Mesh::GetIndexBuffer, which returns the index buffer before it has been committed to the device.
GetDeviceVertexBuffer	Access the mesh's vertex buffer after it has been committed to the device with ID3DX10Mesh::CommitToDevice. This is different from ID3DX10Mesh::GetVertexBuffer, which returns the vertex buffer before it has been committed to the device.
GetFaceCount	Retrieves the number of faces in the mesh.
GetFlags	Access the mesh's creation flags.
GetIndexBuffer	Retrieves the data in an index buffer.
GetPointRepBuffer	Get the mesh's point rep buffer.
GetVertexBuffer	Retrieves the vertex buffer associated with the mesh.
GetVertexBufferCount	Get the number of vertex buffers in the mesh.
GetVertexCount	Get the number of vertices in the mesh. A mesh may contain multiple vertex buffers (i.e. one vertex buffer may contain all position data, another may contains all texture coordinate data, etc.), however each vertex buffer will contain the same number of elements.
GetVertexDescription	Access the vertex description passed into D3DX10CreateMesh. The vertex description describes the layout of the mesh's vertex buffers.
Intersect	Determines if a ray intersects with this mesh.
IntersectSubset	Determines if a ray intersects with a subset of this mesh.

Method	Description
Optimize	Generates a new mesh with reordered faces and vertices to optimize drawing performance.
SetAdjacencyData	Set the mesh's adjacency data.
SetAttributeData	Set the mesh's attribute data.
SetAttributeTable	Sets the attribute table for a mesh and the number of entries stored in the table.
SetIndexData	Set the mesh's index data.
SetPointRepData	Set the point rep data for the mesh.
SetVertexData	Set vertex data into one of the mesh's vertex buffers.

Remarks

To obtain the ID3DX10Mesh interface, call D3DX10CreateMesh.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

D3DX Interfaces

ID3DX10Mesh::CloneMesh method

Article • 03/15/2021 • 2 minutes to read

Creates a new mesh and fills it with the data of a previously loaded mesh.

Syntax

```
C++

HRESULT CloneMesh(
  [in]        UINT                     Flags,
  [in]        LPCSTR                   pPosSemantic,
  [in]  const D3D10_INPUT_ELEMENT_DESC *pDesc,
  [in]        UINT                     DeclCount,
  [out]       ID3DX10Mesh              **ppCloneMesh
);
```

Parameters

Flags [in]

Type: **UINT**

Creation flags to be applied to the new mesh. See **D3DX10_MESH**.

pPosSemantic [in]

Type: **LPCSTR**

The semantic name for the position data.

pDesc [in]

Type: **const D3D10_INPUT_ELEMENT_DESC***

Array of D3D10_INPUT_ELEMENT_DESC structures, describing the vertex format for the returned mesh. See **D3D10_INPUT_ELEMENT_DESC**.

DeclCount [in]

Type: **UINT**

The number of elements in the pDesc array.

ppCloneMesh [out]

Type: **ID3DX10Mesh****

The new mesh.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Mesh

D3DX Interfaces

ID3DX10Mesh::CommitToDevice method

Article • 03/15/2021 • 2 minutes to read

Commit any changes made to a mesh to the device so that the changes can be rendered. This should be called after a mesh's data is altered and before it is rendered. A mesh cannot be rendered unless it is committed to the device. See remarks.

Syntax

```
C++

HRESULT CommitToDevice();
```

Parameters

This method has no parameters.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Remarks

When a mesh is loaded, it's data is loaded into staging resources, meaning the data can be altered but not rendered. When CommitToDevice is called, the data from the staging resources are copied into device resources so that they can be rendered. Although the data is committed to the device, the staging resources remain and can be modified. If any modifications are made to the staging resources, the staging resources must be committed to the device again in order for those changes to be rendered on screen.

Requirements

Requirement	Value
Header	D3DX10.h

Requirement	Value
Library	D3DX10.lib

See also

ID3DX10Mesh

D3DX Interfaces

ID3DX10Mesh::Discard method

Article • 03/15/2021 • 2 minutes to read

Removes mesh data from the device that has been committed to the device (with **ID3DX10Mesh::CommitToDevice**).

Syntax

```
C++

HRESULT Discard(
  [in] D3DX10_MESH_DISCARD_FLAGS dwDiscard
);
```

Parameters

dwDiscard [in]

Type: **D3DX10_MESH_DISCARD_FLAGS**

Specifies which pieces of mesh data to discard from the device. See **D3DX10_MESH_DISCARD_FLAGS**.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Mesh

ID3DX10Mesh::DrawSubset method

Article • 04/27/2021 • 2 minutes to read

Draws a subset of a mesh.

Syntax

```
C++

HRESULT DrawSubset(
  [in] UINT AttribId
);
```

Parameters

AttribId [in]

Type: **UINT**

Specifies which subset of the mesh to draw. This value is used to differentiate faces in a mesh as belonging to one or more attribute groups.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Remarks

An attribute table is used to identify areas of the mesh that need to be drawn with different textures, render states, materials, and so on. In addition, the application can use the attribute table to hide portions of a mesh by not drawing a given attribute identifier (AttribId) when drawing the frame.

Requirements

Requirement	Value

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Mesh

D3DX Interfaces

ID3DX10Mesh::DrawSubsetInstanced method

Article • 05/24/2021 • 2 minutes to read

Draw several instances of the same subset of a mesh.

Syntax

```C++
HRESULT DrawSubsetInstanced(
  [in] UINT AttribId,
  [in] UINT InstanceCount,
  [in] UINT StartInstanceLocation
);
```

Parameters

AttribId [in]

Type: **UINT**

Specifies which subset of the mesh to draw. This value is used to differentiate faces in a mesh as belonging to one or more attribute groups. See remarks.

InstanceCount [in]

Type: **UINT**

Number of instances to render.

StartInstanceLocation [in]

Type: **UINT**

Which instance to start fetching from in each buffer marked as instance data.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Remarks

A mesh contains an attribute table. The attribute table can divide a mesh into subsets, where each subset is identified with an attribute ID. For example, a mesh with 200 faces, divided into three subsets, might have an attribute table that looks like this:

Subset	Faces
AttribID 0	Faces 0 ~ 50
AttribID 1	Faces 51 ~ 125
AttribID 2	Faces 126 ~ 200

Instancing may extend performance by reusing the same geometry to draw multiple objects in a scene. One example of instancing could be to draw the same object with different positions and colors. Indexing requires multiple vertex buffers: at least one for per-vertex data and a second buffer for per-instance data.

Drawing instances with DrawSubsetInstanced is very similar to the process used with **ID3D10Device::DrawIndexedInstanced** that is outlined in Instancing Sample. The key difference when using DrawSubsetInstanced is that vertex and index buffers must be extracted from the **ID3DX10Mesh Interface** object before the instancing data can be combined.

The following code illustrates extracting the vertex and index buffers from the mesh object.

```
ID3D10Buffer* vertexBuffer;
pDeviceObj->pMesh->GetDeviceVertexBuffer(0, &vertexBuffer);
ID3D10Buffer* indexBuffer;
pDeviceObj->pMesh->GetDeviceIndexBuffer(&indexBuffer);
```

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Mesh

D3DX Interfaces

ID3DX10Mesh::GenerateAdjacencyAndPointReps method

Article • 04/27/2021 • 2 minutes to read

Generate a list of mesh edges, as well as a list of faces that share each edge.

Syntax

```
C++

HRESULT GenerateAdjacencyAndPointReps(
  [in] FLOAT Epsilon
);
```

Parameters

Epsilon [in]

Type: **FLOAT**

Specifies that vertices that differ in position by less than epsilon should be treated as coincident.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Remarks

After an application generates adjacency information for a mesh, the mesh data can be optimized for better drawing performance.

The order of the entries in the adjacency buffer is determined by the order of the vertex indices in the index buffer. The adjacent triangle 0 always corresponds to the edge between the indices of the corners 0 and 1. The adjacent triangle 1 always corresponds to the edge between the indices of the corners 1 and 2 while the adjacent triangle 2 corresponds to the edge between the indices of the corners 2 and 0.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Mesh

D3DX Interfaces

ID3DX10Mesh::GenerateAttributeBufferFromTable method

Article • 03/15/2021 • 2 minutes to read

Generate an attribute buffer from the data in the mesh's attribute table. An attribute buffer is another format for storing the data in the attribute table. Both the attribute buffer and the attribute table are internal data structures in the mesh.

Syntax

```
C++

HRESULT GenerateAttributeBufferFromTable();
```

Parameters

This method has no parameters.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Mesh

D3DX Interfaces

ID3DX10Mesh::GenerateGSAdjacency method

Article • 03/15/2021 • 2 minutes to read

Adds adjacency data to the mesh's index buffer. When the mesh is to be sent to a geometry shader that takes adjacency data, it is neccessary for the mesh's index buffer to contain adjacency data.

Syntax

```
C++

HRESULT GenerateGSAdjacency();
```

Parameters

This method has no parameters.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Mesh

D3DX Interfaces

ID3DX10Mesh::GetAdjacencyBuffer method

Article • 03/15/2021 • 2 minutes to read

Access the mesh's adjacency buffer.

Syntax

```
C++

HRESULT GetAdjacencyBuffer(
  [out] ID3DX10MeshBuffer **ppAdjacency
);
```

Parameters

ppAdjacency [out]

Type: **ID3DX10MeshBuffer****

The adjacency buffer. See **ID3DX10MeshBuffer**.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Mesh

ID3DX10Mesh::GetAttributeBuffer method

Article • 03/15/2021 • 2 minutes to read

Access the mesh's attribute buffer.

Syntax

C++

```
HRESULT GetAttributeBuffer(
  [out] ID3DX10MeshBuffer **ppAttributeBuffer
);
```

Parameters

ppAttributeBuffer [out]

Type: **ID3DX10MeshBuffer****

The attribute buffer. See **ID3DX10MeshBuffer**.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Mesh

ID3DX10Mesh::GetAttributeTable method

Article • 04/27/2021 • 2 minutes to read

Retrieves either an attribute table for a mesh, or the number of entries stored in an attribute table for a mesh.

Syntax

```
C++

HRESULT GetAttributeTable(
  [in] D3DX10_ATTRIBUTE_RANGE *pAttribTable,
  [in] UINT                   *pAttribTableSize
);
```

Parameters

pAttribTable [in]

Type: **D3DX10_ATTRIBUTE_RANGE***

Pointer to an array of D3DX10_ATTRIBUTE_RANGE structures, representing the entries in the mesh's attribute table. Specify **NULL** to retrieve the value for pAttribTableSize.

pAttribTableSize [in]

Type: **UINT***

Pointer to either the number of entries stored in pAttribTable or a value to be filled in with the number of entries stored in the attribute table for the mesh.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Remarks

An attribute table is used to identify areas of the mesh that need to be drawn with different textures, render states, materials, and so on. In addition, the application can use the attribute table to hide portions of a mesh by not drawing a given attribute identifier when drawing the frame.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Mesh

D3DX Interfaces

ID3DX10Mesh::GetDeviceIndexBuffer method

Article • 03/15/2021 • 2 minutes to read

Access the mesh's index buffer after it has been committed to the device with ID3DX10Mesh::CommitToDevice. This is different from ID3DX10Mesh::GetIndexBuffer, which returns the index buffer before it has been committed to the device.

Syntax

```
C++

HRESULT GetDeviceIndexBuffer(
  [out] ID3D10Buffer **ppIndexBuffer
);
```

Parameters

ppIndexBuffer [out]

Type: ID3D10Buffer**

The index buffer after it has been committed to the device.

Return value

Type: HRESULT

The return value is one of the values listed in Direct3D 10 Return Codes.

Remarks

If the mesh's index buffer has not already been committed to the device, this API will automatically commit the index buffer before it returns a pointer to the buffer.

Requirements

Requirement	Value

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Mesh

D3DX Interfaces

ID3DX10Mesh::GetDeviceVertexBuffer method

Article • 03/15/2021 • 2 minutes to read

Access the mesh's vertex buffer after it has been committed to the device with ID3DX10Mesh::CommitToDevice. This is different from ID3DX10Mesh::GetVertexBuffer, which returns the vertex buffer before it has been committed to the device.

Syntax

```
C++

HRESULT GetDeviceVertexBuffer(
  [in]  UINT         iBuffer,
  [out] ID3D10Buffer **ppVertexBuffer
);
```

Parameters

iBuffer [in]

Type: UINT

An index identifying which vertex buffer to access.

ppVertexBuffer [out]

Type: ID3D10Buffer**

The vertex buffer after it has been committed to the device.

Return value

Type: HRESULT

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Mesh

D3DX Interfaces

ID3DX10Mesh::GetFaceCount method

Article • 04/27/2021 • 2 minutes to read

Retrieves the number of faces in the mesh.

Syntax

```
C++

UINT GetFaceCount();
```

Parameters

This method has no parameters.

Return value

Type: **UINT**

Returns the number of faces in the mesh.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Mesh

D3DX Interfaces

ID3DX10Mesh::GetFlags method

Article • 03/15/2021 • 2 minutes to read

Access the mesh's creation flags.

Syntax

```
C++

UINT GetFlags();
```

Parameters

This method has no parameters.

Return value

Type: **UINT**

The creation flags passed into the *options* parameter of **D3DX10CreateMesh** when the mesh was created. See **D3DX10_MESH**.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Mesh

D3DX Interfaces

ID3DX10Mesh::GetIndexBuffer method

Article • 04/27/2021 • 2 minutes to read

Retrieves the data in an index buffer.

Syntax

```
C++

HRESULT GetIndexBuffer(
  [out] ID3DX10MeshBuffer **ppIndexBuffer
);
```

Parameters

ppIndexBuffer [out]

Type: **ID3DX10MeshBuffer****

Address of a pointer to a ID3DX10MeshBuffer interface, representing the index buffer associated with the mesh.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Mesh

ID3DX10Mesh::GetPointRepBuffer method

Article • 03/15/2021 • 2 minutes to read

Get the mesh's point rep buffer.

Syntax

```
C++

HRESULT GetPointRepBuffer(
  [out] ID3DX10MeshBuffer **ppPointReps
);
```

Parameters

ppPointReps [out]

Type: **ID3DX10MeshBuffer****

Pointer to a mesh buffer containing the mesh's point rep data. See **ID3DX10MeshBuffer**.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Mesh

ID3DX10Mesh::GetVertexBuffer method

Article • 04/27/2021 • 2 minutes to read

Retrieves the vertex buffer associated with the mesh.

Syntax

```C++
HRESULT GetVertexBuffer(
  [in]  UINT              iBuffer,
  [out] ID3DX10MeshBuffer **ppVertexBuffer
);
```

Parameters

iBuffer [in]

Type: **UINT**

The vertex buffer to get. This is an index value.

ppVertexBuffer [out]

Type: **ID3DX10MeshBuffer****

The vertex buffer. See **ID3DX10MeshBuffer**

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Mesh

D3DX Interfaces

ID3DX10Mesh::GetVertexBufferCount method

Article • 03/15/2021 • 2 minutes to read

Get the number of vertex buffers in the mesh.

Syntax

```
C++

UINT GetVertexBufferCount();
```

Parameters

This method has no parameters.

Return value

Type: **UINT**

The number of vertex buffers in the mesh.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Mesh

D3DX Interfaces

ID3DX10Mesh::GetVertexCount method

Article • 03/15/2021 • 2 minutes to read

Get the number of vertices in the mesh. A mesh may contain multiple vertex buffers (i.e. one vertex buffer may contain all position data, another may contains all texture coordinate data, etc.), however each vertex buffer will contain the same number of elements.

Syntax

```
C++

UINT GetVertexCount();
```

Parameters

This method has no parameters.

Return value

Type: **UINT**

The number of vertices in the mesh.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Mesh

D3DX Interfaces

ID3DX10Mesh::GetVertexDescription method

Article • 03/15/2021 • 2 minutes to read

Access the vertex description passed into D3DX10CreateMesh. The vertex description describes the layout of the mesh's vertex buffers.

Syntax

```
C++

HRESULT GetVertexDescription(
  [out] const D3D10_INPUT_ELEMENT_DESC **ppDesc,
  [in]        UINT                     *pDeclCount
);
```

Parameters

ppDesc [out]

Type: **const** D3D10_INPUT_ELEMENT_DESC**

Array of input elements that describe the layout of the mesh's vertex buffers. See D3D10_INPUT_ELEMENT_DESC.

pDeclCount [in]

Type: UINT*

The number of input elements in ppDesc.

Return value

Type: HRESULT

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Mesh

D3DX Interfaces

ID3DX10Mesh::Intersect method

Article • 03/15/2021 • 2 minutes to read

Determines if a ray intersects with this mesh.

Syntax

C++

```
HRESULT Intersect(
  [in]  D3DXVECTOR3 *pRayPos,
  [in]  D3DXVECTOR3 *pRayDir,
  [in]  UINT        *pHitCount,
  [in]  UINT        *pFaceIndex,
  [in]  float       *pU,
  [in]  float       *pV,
  [in]  float       *pDist,
  [out] ID3D10Blob  **ppAllHits
);
```

Parameters

pRayPos [in]

Type: **D3DXVECTOR3***

Pointer to a **D3DXVECTOR3** structure, specifying the point where the ray begins.

pRayDir [in]

Type: **D3DXVECTOR3***

Pointer to a **D3DXVECTOR3** structure, specifying the direction of the ray.

pHitCount [in]

Type: **UINT***

The number of times the ray intersected with the mesh.

pFaceIndex [in]

Type: **UINT***

Pointer to an index value of the face closest to the ray origin, if pHit is **TRUE**.

pU [in]

Type: **float***

Pointer to a barycentric hit coordinate, U.

pV [in]

Type: **float***

Pointer to a barycentric hit coordinate, V.

pDist [in]

Type: **float***

Pointer to a ray intersection parameter distance.

ppAllHits [out]

Type: **ID3D10Blob****

Pointer to an **ID3D10Blob Interface**, containing an array of **D3DX10_INTERSECT_INFO** structures. This is a list of all the hits that occurred in the intersection test.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Remarks

This API provides a way to understand points in and around a triangle, independent of where the triangle is actually located. This function returns the resulting point by using the following equation: V1 + U(V2 - V1) + V(V3 - V1).

Any point in the plane V1V2V3 can be represented by the barycentric coordinate (U,V). The parameter U controls how much V2 gets weighted into the result, and the parameter V controls how much V3 gets weighted into the result. Lastly, the value of [1 - (U + V)] controls how much V1 gets weighted into the result.

Barycentric coordinates are a form of general coordinates. In this context, using barycentric coordinates represents a change in coordinate systems. What holds true for Cartesian coordinates holds true for barycentric coordinates.

Barycentric coordinates define a point inside a triangle in terms of the triangle's vertices. For a more in-depth description of barycentric coordinates, see Mathworld's Barycentric Coordinates Description.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Mesh

D3DX Interfaces

ID3DX10Mesh::IntersectSubset method

Article • 03/15/2021 • 2 minutes to read

Determines if a ray intersects with a subset of this mesh.

Syntax

C++

```
HRESULT IntersectSubset(
  [in]  UINT        AttribId,
  [in]  D3DXVECTOR3 *pRayPos,
  [in]  D3DXVECTOR3 *pRayDir,
  [in]  UINT        *pHitCount,
  [in]  UINT        *pFaceIndex,
  [in]  float       *pU,
  [in]  float       *pV,
  [in]  float       *pDist,
  [out] ID3D10Blob  **ppAllHits
);
```

Parameters

AttribId [in]

Type: **UINT**

Attribute ID identifying the subset of the mesh.

pRayPos [in]

Type: **D3DXVECTOR3***

Pointer to a **D3DXVECTOR3** structure, specifying the point where the ray begins.

pRayDir [in]

Type: **D3DXVECTOR3***

Pointer to a **D3DXVECTOR3** structure, specifying the direction of the ray.

pHitCount [in]

Type: **UINT***

The number of times the ray intersected with the mesh.

pFaceIndex [in]

Type: **UINT***

Pointer to an index value of the face closest to the ray origin, if pHit is **TRUE**.

pU [in]

Type: **float***

Pointer to a barycentric hit coordinate, U.

pV [in]

Type: **float***

Pointer to a barycentric hit coordinate, V.

pDist [in]

Type: **float***

Pointer to a ray intersection parameter distance.

ppAllHits [out]

Type: **ID3D10Blob****

Pointer to an **ID3D10Blob Interface**, containing an array of **D3DX10_INTERSECT_INFO** structures. This is a list of all the hits that occurred in the intersection test.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Remarks

This API provides a way to understand points in and around a triangle, independent of where the triangle is actually located. This function returns the resulting point by using the following equation: V1 + U(V2 - V1) + V(V3 - V1).

Any point in the plane V1V2V3 can be represented by the barycentric coordinate (U,V). The parameter U controls how much V2 gets weighted into the result, and the parameter V controls how much V3 gets weighted into the result. Lastly, the value of [1 - (U + V)] controls how much V1 gets weighted into the result.

Barycentric coordinates are a form of general coordinates. In this context, using barycentric coordinates represents a change in coordinate systems. What holds true for Cartesian coordinates holds true for barycentric coordinates.

Barycentric coordinates define a point inside a triangle in terms of the triangle's vertices. For a more in-depth description of barycentric coordinates, see Mathworld's Barycentric Coordinates Description.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Mesh

D3DX Interfaces

ID3DX10Mesh::Optimize method

Article • 04/27/2021 • 2 minutes to read

Generates a new mesh with reordered faces and vertices to optimize drawing performance.

Syntax

```
C++

HRESULT Optimize(
  [in]  UINT        Flags,
  [in]  UINT        *pFaceRemap,
  [out] LPD3D10BLOB *ppVertexRemap
);
```

Parameters

Flags [in]

Type: **UINT**

Specifies the type of optimization to perform. This parameter can be set to a combination of one or more flags from D3DXMESHOPT and D3DXMESH (except D3DXMESH_32BIT, D3DXMESH_IB_WRITEONLY, and D3DXMESH_WRITEONLY).

pFaceRemap [in]

Type: **UINT***

An array of UINTs, one per face, that identifies the original mesh face that corresponds to each face in the optimized mesh. If the value supplied for this argument is **NULL**, face remap data is not returned.

ppVertexRemap [out]

Type: **LPD3D10BLOB***

Address of a pointer to an **ID3D10Blob Interface**, which contains a DWORD for each vertex that specifies how the new vertices map to the old vertices. This remap is useful if you need to alter external data based on the new vertex mapping.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Remarks

This method generates a new mesh. Before running Optimize, an application must generate an adjacency buffer by calling **ID3DX10Mesh::GenerateAdjacencyAndPointReps**. The adjacency buffer contains adjacency data, such as a list of edges and the faces that are adjacent to each other.

This method is very similar to the **ID3DX10Mesh::CloneMesh** method, except that it can perform optimization while generating the new clone of the mesh. The output mesh inherits all of the creation parameters of the input mesh.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Mesh

D3DX Interfaces

ID3DX10Mesh::SetAdjacencyData method

Article • 03/15/2021 • 2 minutes to read

Set the mesh's adjacency data.

Syntax

```
C++

HRESULT SetAdjacencyData(
  [in] const UINT *pAdjacency
);
```

Parameters

pAdjacency [in]

Type: **const UINT***

The adjacency data to set.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Mesh

ID3DX10Mesh::SetAttributeData method

Article • 03/15/2021 • 2 minutes to read

Set the mesh's attribute data.

Syntax

```
C++

HRESULT SetAttributeData(
  [in] const UINT *pData
);
```

Parameters

pData [in]

Type: **const UINT***

The attribute data to set.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Mesh

D3DX Interfaces

ID3DX10Mesh::SetAttributeTable method

Article • 04/27/2021 • 2 minutes to read

Sets the attribute table for a mesh and the number of entries stored in the table.

Syntax

```
C++

HRESULT SetAttributeTable(
  [in] const D3DX10_ATTRIBUTE_RANGE *pAttribTable,
  [in]       UINT                   cAttribTableSize
);
```

Parameters

pAttribTable [in]

Type: **const** D3DX10_ATTRIBUTE_RANGE*

Pointer to an array of D3DX10_ATTRIBUTE_RANGE structures, representing the entries in the mesh attribute table.

cAttribTableSize [in]

Type: UINT

Number of attributes in the mesh attribute table.

Return value

Type: HRESULT

The return value is one of the values listed in Direct3D 10 Return Codes.

Remarks

If an application keeps track of the information in an attribute table, and rearranges the table as a result of changes to attributes or faces, this method allows the application to

update the attribute tables instead of calling ID3DX10Mesh::Optimize again.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Mesh

D3DX Interfaces

ID3DX10Mesh::SetIndexData method

Article • 03/15/2021 • 2 minutes to read

Set the mesh's index data.

Syntax

```
C++

HRESULT SetIndexData(
  [in] const void *pData,
  [in]       UINT cIndices
);
```

Parameters

pData [in]

Type: **const void***

The index data.

cIndices [in]

Type: **UINT**

The number of indices in pData.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Mesh

D3DX Interfaces

ID3DX10Mesh::SetPointRepData method

Article • 03/15/2021 • 2 minutes to read

Set the point rep data for the mesh.

Syntax

```
C++

HRESULT SetPointRepData(
  [in] const UINT *pPointReps
);
```

Parameters

pPointReps [in]

Type: **const UINT***

The point rep data to set.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Mesh

D3DX Interfaces

ID3DX10Mesh::SetVertexData method

Article • 03/15/2021 • 2 minutes to read

Set vertex data into one of the mesh's vertex buffers.

Syntax

```
C++

HRESULT SetVertexData(
  [in]       UINT iBuffer,
  [in] const void *pData
);
```

Parameters

iBuffer [in]

Type: **UINT**

The vertex buffer to be filled with pData.

pData [in]

Type: **const void***

The vertex data to set.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Mesh

D3DX Interfaces

ID3DX10MeshBuffer interface

Article • 03/15/2021 • 2 minutes to read

A mesh buffer is a buffer that contains data about a mesh. It can contain one of five different types of data: vertex data, index data, adjacency data, attribute data, or point rep data. The structure is used to access these five pieces of data through the following five APIs: ID3DX10Mesh::GetVertexBuffer, ID3DX10Mesh::GetIndexBuffer, ID3DX10Mesh::GetAdjacencyBuffer, ID3DX10Mesh::GetAttributeBuffer, or ID3DX10Mesh::GetPointRepBuffer.

Members

The **ID3DX10MeshBuffer** interface inherits from the IUnknown interface. **ID3DX10MeshBuffer** also has these types of members:

- Methods

Methods

The **ID3DX10MeshBuffer** interface has these methods.

Method	Description
GetSize	Get the size of the mesh buffer, in bytes.
Map	Get a pointer to the mesh buffer memory to modify its contents.
Unmap	Unmap a buffer.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10MeshBuffer::GetSize method

Article • 03/15/2021 • 2 minutes to read

Get the size of the mesh buffer, in bytes.

Syntax

```
C++

SIZE_T GetSize();
```

Parameters

This method has no parameters.

Return value

Type: **SIZE_T**

The size of the mesh buffer, in bytes.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10MeshBuffer

D3DX Interfaces

ID3DX10MeshBuffer::Map method

Article • 05/24/2021 • 2 minutes to read

Get a pointer to the mesh buffer memory to modify its contents.

Syntax

```
C++

HRESULT Map(
  [out] void   **ppData,
  [out] SIZE_T *pSize
);
```

Parameters

ppData [out]

Type: **void****

Pointer to the buffer data.

pSize [out]

Type: **SIZE_T***

Size of the buffer in bytes.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Remarks

Differences between Direct3D 9 and Direct3D 10:

- Map() in Direct3D 10 is analogous to resource Map() in Direct3D 9.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10MeshBuffer

D3DX Interfaces

ID3DX10MeshBuffer::Unmap method

Article • 05/24/2021 • 2 minutes to read

Unmap a buffer.

Syntax

```
C++

HRESULT Unmap();
```

Parameters

This method has no parameters.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Remarks

Differences between Direct3D 9 and Direct3D 10:

- Unmap() in Direct3D 10 is analogous to resource Unlock() in Direct3D 9.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10MeshBuffer

D3DX Interfaces

ID3DX10SkinInfo interface

Article • 03/15/2021 • 2 minutes to read

ID3DX10SkinInfo allows you to optimize, process, and manually set the relationship between bones and vertices in your meshes (see Skeletal Animation on Wikipedia). It is most useful for making .x files exported by DCC Apps (such as 3DS Max and Maya) more hardware-friendly, and for improving the render speed of your skinned meshes in software render mode.

Members

The **ID3DX10SkinInfo** interface inherits from the **IUnknown** interface. **ID3DX10SkinInfo** also has these types of members:

- Methods

Methods

The **ID3DX10SkinInfo** interface has these methods.

Method	Description
AddBoneInfluences	Enable an existing bone to influence a group of vertices and define how much influence the bone has on each vertex.
AddBones	Allocate space for more bones.
AddVertices	Allocate space for additional vertices.
ClearBoneInfluences	Clear a bone's list of vertices that it influences.
Compact	Limit the number of bones that can influence a vertex and/or limit the amount of influence a bone can have on a vertex.
DoSoftwareSkinning	Do software skinning on an array of vertices.
FindBoneInfluenceIndex	Find the index that indicates where a given vertex is in a given bone's list of influenced vertices.
GetBoneInfluence	Get the amount of influence a given bone has over a given vertex.
GetBoneInfluenceCount	Get the number of vertices that a given bone influences.
GetBoneInfluences	Get a list of vertices that a given bone influences and a list of the amount of influence that bone has on each vertex.

Method	Description
GetMaxBoneInfluences	Get the number of vertices a bone can maximally influence.
GetNumBones	Get the number of bones in ID3DX10SkinInfo.
GetNumVertices	Get the number of vertices in ID3DX10SkinInfo.
RemapBones	Change which bones influence which vertices.
RemapVertices	Change which vertices are influenced by which bones.
RemoveBone	Remove a bone.
SetBoneInfluence	Set the amount of influence a given bone has over a given vertex.

Remarks

Create a ID3DX10SkinInfo interface with D3DX10CreateSkinInfo, **D3DX10CreateSkinInfoFromBlendedMesh**, or **D3DX10CreateSkinInfoFVF**.

The LPD3DX10SKININFO type is defined as a pointer to the **ID3DX10SkinInfo** interface.

```
typedef struct ID3DX10SkinInfo *LPD3DX10SKININFO;
```

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

D3DX Interfaces

ID3DX10SkinInfo::AddBoneInfluences method

Article • 03/15/2021 • 2 minutes to read

Enable an existing bone to influence a group of vertices and define how much influence the bone has on each vertex.

Syntax

```C++
HRESULT AddBoneInfluences(
  [in] UINT  BoneIndex,
  [in] UINT  InfluenceCount,
  [in] UINT  *pIndices,
  [in] float *pWeights
);
```

Parameters

BoneIndex [in]

Type: **UINT**

An index that specifies an existing bone. Must be between 0 and the value returned by **ID3DX10SkinInfo::GetNumBones**.

InfluenceCount [in]

Type: **UINT**

Number of vertices to add to the bone's influence.

pIndices [in]

Type: **UINT***

Pointer to an array of vertex indices. Each member of this array has a corresponding member in pWeights, such that pIndices[i] corresponds to pWeights[i]. The corresponding value in pWeights[i] determines how much influence BoneIndex will have on the vertex indexed by pIndices[i]. The size of the pIndices array must be equal to or greater than InfluenceCount.

pWeights [in]

Type: **float***

Pointer to an array of bone weights. Each member of this array has a corresponding member in pIndices, such that pWeights[i] corresponds to pIndices[i]. Each value in pWeights is between 0 and 1 and defines the amount of influence the bone has over each vertex. The size of pWeights must be equal to or greater than InfluenceCount.

Return value

Type: **HRESULT**

If the method succeeds, the return value is S_OK. If the method fails, the return value can be: E_INVALIDARG or E_OUTOFMEMORY.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10SkinInfo

D3DX Interfaces

ID3DX10SkinInfo::AddBones method

Article • 03/15/2021 • 2 minutes to read

Allocate space for more bones.

Syntax

```C++
HRESULT AddBones(
  [in] UINT Count
);
```

Parameters

Count [in]

Type: UINT

The number of bones to add.

Return value

Type: HRESULT

If this method succeeds, the return value is S_OK. If the method fails, the return value can be: E_OUTOFMEMORY.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10SkinInfo

ID3DX10SkinInfo::AddVertices method

Article • 03/15/2021 • 2 minutes to read

Allocate space for additional vertices.

Syntax

```
C++

HRESULT AddVertices(
  [in] UINT Count
);
```

Parameters

Count [in]

Type: **UINT**

The number of vertices to add.

Return value

Type: **HRESULT**

If this method succeeds, the return value is S_OK. If the method fails, the return value can be: E_OUTOFMEMORY.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10SkinInfo

ID3DX10SkinInfo::ClearBoneInfluences method

Article • 03/15/2021 • 2 minutes to read

Clear a bone's list of vertices that it influences.

Syntax

```
C++

HRESULT ClearBoneInfluences(
  [in] UINT BoneIndex
);
```

Parameters

BoneIndex [in]

Type: UINT

An index that specifies an existing bone. Must be between 0 and the value returned by ID3DX10SkinInfo::GetNumBones.

Return value

Type: HRESULT

If the method succeeds, the return value is S_OK. If the method fails, the return value can be: E_INVALIDARG.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10SkinInfo

D3DX Interfaces

ID3DX10SkinInfo::Compact method

Article • 03/15/2021 • 2 minutes to read

Limit the number of bones that can influence a vertex and/or limit the amount of influence a bone can have on a vertex.

Syntax

```C++
HRESULT Compact(
  [in] UINT  MaxPerVertexInfluences,
  [in] UINT  ScaleMode,
  [in] float MinWeight
);
```

Parameters

MaxPerVertexInfluences [in]

Type: **UINT**

The maximum number of bones that can influence any given vertex. This value is ignored if it is greater than the value returned by **ID3DX10SkinInfo::GetMaxBoneInfluences**.

ScaleMode [in]

Type: **UINT**

A flag describing how to scale the remaining weights on a given vertex after some have been chopped off by MinWeight. If D3DX10_SKININFO_NO_SCALING is specified, the weights will not be scaled at all. If D3DX10_SKININFO_SCALE_TO_1 is specified, the weights greater than MinWeight will be scaled up so that they add up to 1.0. If D3DX10_SKININFO_SCALE_TO_TOTAL is specified, the weights greater than MinWeight will be scaled up so that they add up to the original total.

MinWeight [in]

Type: **float**

The minimum percentage of influence, or weight, that any bone can have on any vertex. This value must be between 0 and 1.

Return value

Type: **HRESULT**

If the method succeeds, the return value is S_OK. If the method fails, the return value can be: E_OUTOFMEMORY or E_INVALIDARG.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10SkinInfo

D3DX Interfaces

ID3DX10SkinInfo::DoSoftwareSkinning method

Article • 03/15/2021 • 2 minutes to read

Do software skinning on an array of vertices.

Syntax

```C++
HRESULT DoSoftwareSkinning(
  [in]      UINT                    StartVertex,
  [in]      UINT                    VertexCount,
  [in]      void                    *pSrcVertices,
  [in]      UINT                    SrcStride,
  [in, out] void                    *pDestVertices,
  [in]      UINT                    DestStride,
  [in]      D3DXMATRIX              *pBoneMatrices,
  [in]      D3DXMATRIX              *pInverseTransposeBoneMatrices,
  [in]      D3DX10_SKINNING_CHANNEL *pChannelDescs,
  [in]      UINT                    NumChannels
);
```

Parameters

StartVertex [in]

Type: **UINT**

A 0-based index into pSrcVertices.

VertexCount [in]

Type: **UINT**

Number of vertices to transform.

pSrcVertices [in]

Type: **void***

Pointer to an array of vertices to transform.

SrcStride [in]

Type: **UINT**

The size, in bytes, of a vertex in pSrcVertices.

pDestVertices [in, out]

Type: **void***

Pointer to an array of vertices, which will be filled with the transformed vertices.

DestStride [in]

Type: **UINT**

The size, in bytes, of a vertex in pDestVertices.

pBoneMatrices [in]

Type: **D3DXMATRIX***

An array of matrices that will be used to transform the points mapped to each bone, such that the vertices mapped to bone[i] will be transformed by pBoneMatrices[i]. This array will be used to transform the matrices only if the IsNormal value in pChannelDescs is set to **FALSE**, otherwise pInverseTransposeBoneMatrices will be used.

pInverseTransposeBoneMatrices [in]

Type: **D3DXMATRIX***

If this value is **NULL**, it will be set equal to pBoneMatrices. This array of matrices will be used to transform the vertices only if the IsNormal value in pChannelDescs is set to **TRUE**, otherwise pBoneMatrices will be used.

pChannelDescs [in]

Type: **D3DX10_SKINNING_CHANNEL***

Pointer to a D3DX10_SKINNING_CHANNEL structure, which determines the member of the vertex decl the software skinning will be done on.

NumChannels [in]

Type: **UINT**

The number of D3DX10_SKINNING_CHANNEL structures in pChannelDescs.

Return value

Type: HRESULT

If the method succeeds, the return value is S_OK. If the method fails, the return value can be: E_INVALIDARG.

Remarks

Here is an example of how to use software skinning:

```
//vertex definition
struct MyVertex
{
    D3DXVECTOR3 Position;
    D3DXVECTOR2 Weight;
    D3DXVECTOR2 TexCoord;
};

//create vertex data
const UINT numVertices = 16;
MyVertex vertices[numVertices] = {...};
MyVertex destVertices[numVertices];

//create bone matrices
D3DXMATRIX boneMatrices[2];
D3DXMatrixIdentity(&boneMatrices[0]);
D3DXMatrixRotationX(&boneMatrices[1], 3.14159f / 180.0f);

//create bone indices and weights
UINT boneIndices[numVertices] = {...};
float boneWeights[2][numVertices] = {...};

//create skin info, populate it with bones and vertices, and then map them
to each other
ID3DX10SkinInfo *pSkinInfo = NULL;
D3DX10CreateSkinInfo(&pSkinInfo);
pSkinInfo->AddBones(2);
pSkinInfo->AddVertices(numVertices);
pSkinInfo->AddBoneInfluences(0, numVertices, boneIndices, boneWeights[0]);
pSkinInfo->AddBoneInfluences(1, numVertices, boneIndices, boneWeights[1]);

//create channel desc
D3DX10_SKINNING_CHANNEL channelDesc;
channelDesc.SrcOffset = 0;
channelDesc.DestOffset = 0;
channelDesc.IsNormal = FALSE;

//do the skinning
pSkinInfo->DoSoftwareSkinning(0, numVertices,
                              vertices, sizeof(MyVertex),
                              destVertices, sizeof(MyVertex),
```

```
                                boneMatrices, NULL,
                                &channelDesc, 1);
```

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10SkinInfo

D3DX Interfaces

ID3DX10SkinInfo::FindBoneInfluenceIndex method

Article • 03/15/2021 • 2 minutes to read

Find the index that indicates where a given vertex is in a given bone's list of influenced vertices.

Syntax

```
C++

HRESULT FindBoneInfluenceIndex(
  [in] UINT BoneIndex,
  [in] UINT VertexIndex,
  [in] UINT *pInfluenceIndex
);
```

Parameters

BoneIndex [in]

Type: **UINT**

An index that specifies an existing bone. Must be between 0 and the value returned by **ID3DX10SkinInfo::GetNumBones**.

VertexIndex [in]

Type: **UINT**

The index of the vertex in the vertex buffer.

pInfluenceIndex [in]

Type: **UINT***

The index of the vertex in the bone's list of influenced vertices.

Return value

Type: **HRESULT**

If the method succeeds, the return value is S_OK. If the method fails, the return value can be: E_INVALIDARG.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10SkinInfo

D3DX Interfaces

ID3DX10SkinInfo::GetBoneInfluence method

Article • 03/15/2021 • 2 minutes to read

Get the amount of influence a given bone has over a given vertex.

Syntax

C++

```
HRESULT GetBoneInfluence(
  [in] UINT  BoneIndex,
  [in] UINT  InfluenceIndex,
  [in] float *pWeight
);
```

Parameters

BoneIndex [in]

Type: **UINT**

An index that specifies an existing bone. Must be between 0 and the value returned by **ID3DX10SkinInfo::GetNumBones**.

InfluenceIndex [in]

Type: **UINT**

An index into the bone's list of vertices that it influences.

pWeight [in]

Type: **float***

The amount of influence, between 0 and 1, that the bone has over the vertex.

Return value

Type: **HRESULT**

If the method succeeds, the return value is D3D_OK. If the method fails, the return value can be E_INVALIDARG.

Remarks

Use ID3DX10SkinInfo::GetBoneInfluenceCount to find out how many vertices the bone influences.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10SkinInfo

D3DX Interfaces

ID3DX10SkinInfo::GetBoneInfluenceCount method

Article • 03/15/2021 • 2 minutes to read

Get the number of vertices that a given bone influences.

Syntax

```
C++

UINT GetBoneInfluenceCount(
  [in] UINT BoneIndex
);
```

Parameters

BoneIndex [in]

Type: UINT

An index that specifies an existing bone. Must be between 0 and the value returned by ID3DX10SkinInfo::GetNumBones.

Return value

Type: UINT

If the method succeeds, the return value is S_OK. If the method fails, the return value can be: E_INVALIDARG.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10SkinInfo

D3DX Interfaces

ID3DX10SkinInfo::GetBoneInfluences method

Article • 03/15/2021 • 2 minutes to read

Get a list of vertices that a given bone influences and a list of the amount of influence that bone has on each vertex.

Syntax

```C++
HRESULT GetBoneInfluences(
  [in]      UINT  BoneIndex,
  [in]      UINT  Offset,
  [in]      UINT  Count,
  [in, out] UINT  *pDestIndices,
  [in, out] float *pDestWeights
);
```

Parameters

BoneIndex [in]

Type: **UINT**

An index that specifies an existing bone. Must be between 0 and the value returned by **ID3DX10SkinInfo::GetNumBones**.

Offset [in]

Type: **UINT**

An offset from the top of the bone's list of influenced vertices. This must be between 0 and the value returned by **ID3DX10SkinInfo::GetBoneInfluenceCount**.

Count [in]

Type: **UINT**

The number of indices and weights to retrieve. Must be between 0 and the value returned by ID3DX10SkinInfo::GetBoneInfluenceCount.

pDestIndices [in, out]

Type: **UINT***

A list of indices into the vertex buffer, each one representing a vertex influenced by the bone. These values correspond to the values in pDestWeights, such that pDestIndices[i] corresponds to pDestWeights[i].

pDestWeights [in, out]

Type: **float***

A list of the amount of influence the bone has on each vertex. These values correspond to the values in pDestIndices, such that pDestWeights[i] corresponds to pDestIndices[i].f

Return value

Type: **HRESULT**

If the method succeeds, the return value is S_OK. If the method fails, the return value can be: E_INVALIDARG or E_OUTOFMEMORY.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10SkinInfo

D3DX Interfaces

ID3DX10SkinInfo::GetMaxBoneInfluences method

Article • 03/15/2021 • 2 minutes to read

Get the number of vertices a bone can maximally influence.

Syntax

```
C++

UINT GetMaxBoneInfluences();
```

Parameters

This method has no parameters.

Return value

Type: **UINT**

The number of vertices a bone can maximally influence.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10SkinInfo

D3DX Interfaces

ID3DX10SkinInfo::GetNumBones method

Article • 03/15/2021 • 2 minutes to read

Get the number of bones in ID3DX10SkinInfo.

Syntax

```
C++

UINT GetNumBones();
```

Parameters

This method has no parameters.

Return value

Type: **UINT**

The number of bones in ID3DX10SkinInfo.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10SkinInfo

D3DX Interfaces

ID3DX10SkinInfo::GetNumVertices method

Article • 03/15/2021 • 2 minutes to read

Get the number of vertices in ID3DX10SkinInfo.

Syntax

```
C++

UINT GetNumVertices();
```

Parameters

This method has no parameters.

Return value

Type: **UINT**

The number of vertices in ID3DX10SkinInfo.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10SkinInfo

D3DX Interfaces

ID3DX10SkinInfo::RemapBones method

Article • 03/15/2021 • 2 minutes to read

Change which bones influence which vertices.

Syntax

```
C++

HRESULT RemapBones(
  [in] UINT NewBoneCount,
  [in] UINT *pBoneRemap
);
```

Parameters

NewBoneCount [in]

Type: **UINT**

The new number of bones.

pBoneRemap [in]

Type: **UINT***

A pointer to an array of bone indices, which describe the remapping. For example, say SkinInfo contains some bones such that bone0 is mapped to v0, bone1 to v1, and bone2 to v2, and array with 2,1,0 is specified for pBoneRemap. This will cause bone0 to be mapped to v2, bone1 to v1, and bone2 to v0.

Return value

Type: **HRESULT**

If the method succeeds, the return value is S_OK. If the method fails, the return value can be: E_OUTOFMEMORY or E_INVALIDARG.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10SkinInfo

D3DX Interfaces

ID3DX10SkinInfo::RemapVertices method

Article • 03/15/2021 • 2 minutes to read

Change which vertices are influenced by which bones.

Syntax

```
C++

HRESULT RemapVertices(
  [in] UINT NewVertexCount,
  [in] UINT *pVertexRemap
);
```

Parameters

NewVertexCount [in]

Type: **UINT**

The new number of vertices.

pVertexRemap [in]

Type: **UINT***

A pointer to an array of vertex indices, which describe the remapping. For example, say SkinInfo contains some vertices such that bone0 is mapped to v0, bone1 to v1, and bone2 to v2, and array with 2,1,0 is specified for pBoneRemap. This will cause bone0 to be mapped to v2, bone1 to v1, and bone2 to v0.

Return value

Type: **HRESULT**

If the method succeeds, the return value is S_OK. If the method fails, the return value can be: E_OUTOFMEMORY or E_INVALIDARG.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10SkinInfo

D3DX Interfaces

ID3DX10SkinInfo::RemoveBone method

Article • 03/15/2021 • 2 minutes to read

Remove a bone.

Syntax

```
C++

HRESULT RemoveBone(
  [in] UINT Index
);
```

Parameters

Index [in]

Type: **UINT**

An index that specifies which bone to remove. Must be between 0 and the value returned by **ID3DX10SkinInfo::GetNumBones**.

Return value

Type: **HRESULT**

If the method succeeds, the return value is S_OK. If the method fails, the return value can be: E_INVALIDARG.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10SkinInfo

ID3DX10SkinInfo::SetBoneInfluence method

Article • 03/15/2021 • 2 minutes to read

Set the amount of influence a given bone has over a given vertex.

Syntax

C++

```
HRESULT SetBoneInfluence(
  [in] UINT  BoneIndex,
  [in] UINT  InfluenceIndex,
  [in] float Weight
);
```

Parameters

BoneIndex [in]

Type: **UINT**

An index that specifies an existing bone. Must be between 0 and the value returned by **ID3DX10SkinInfo::GetNumBones**.

InfluenceIndex [in]

Type: **UINT**

An index into the bone's list of vertices that it influences.

Weight [in]

Type: **float**

The amount of influence, between 0 and 1, that the bone has over the vertex.

Return value

Type: **HRESULT**

If the method succeeds, the return value is D3D_OK. If the method fails, the return value can be E_INVALIDARG.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10SkinInfo

D3DX Interfaces

ID3DX10Sprite interface

Article • 03/15/2021 • 2 minutes to read

The ID3DX10Sprite interface provides a set of methods that simplify the process of drawing sprites using Microsoft Direct3D. This interface can operate on a set of many sprites.

Members

The **ID3DX10Sprite** interface inherits from the **IUnknown** interface. **ID3DX10Sprite** also has these types of members:

- Methods

Methods

The **ID3DX10Sprite** interface has these methods.

Method	Description
Begin	Prepare a device for drawing sprites.
DrawSpritesBuffered	Add an array of sprites to the batch of sprites to be rendered. This must be called in between calls to **ID3DX10Sprite::Begin** and **ID3DX10Sprite::End**, and **ID3DX10Sprite::Flush** must be called before End to send all of the batched sprites to the device for rendering. This draw method is most useful when drawing a small number of sprites that you want buffered into a large batch, such as fonts.
DrawSpritesImmediate	Draw an array of sprites. This will immediately send the sprites to the device for rendering, which is different from **ID3DX10Sprite::DrawSpritesBuffered** which only adds an array of sprites to a batch of sprites to be rendered when **ID3DX10Sprite::Flush** is called. This draw method is most useful when drawing a large number of sprites that have already been sorted on the CPU (or do not need to be sorted), such as in a particle system. This must be called in between calls to **ID3DX10Sprite::Begin** and **ID3DX10Sprite::End**.
End	Call this after ID3DX10Sprite::Flush. If D3DX10_SPRITE_SAVE_STATE was specified when ID3DX10Sprite::Begin was called, this API will restore the device state to how it was before ID3DX10Sprite::Begin was called.
Flush	Force all batched sprites to be submitted to the device. Device states remain as they were after the last call to ID3DX10Sprite::Begin. The list of batched sprites is then cleared.

Method	Description
GetDevice	Retrieve the device associated with the sprite object.
GetProjectionTransform	Get the sprite projection matrix that is applied to all sprites.
GetViewTransform	Get the view transform that applies to all sprites.
SetProjectionTransform	Set the projection matrix for all sprites.
SetViewTransform	Set the view transform that applies to all sprites.

Remarks

The ID3DX10Sprite interface is obtained by calling the **D3DX10CreateSprite** function.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

D3DX Interfaces

ID3DX10Sprite::Begin method

Article • 03/15/2021 • 2 minutes to read

Prepare a device for drawing sprites.

Syntax

```
C++

HRESULT Begin(
  [in] UINT flags
);
```

Parameters

flags [in]

Type: **UINT**

Flags that control how the sprites will be drawn. See **D3DX10_SPRITE_FLAG**.

Return value

Type: **HRESULT**

If the method succeeds, the return value is S_OK. If the method fails, the return value can be one of the following: D3DERR_INVALIDCALL, D3DERR_OUTOFVIDEOMEMORY, D3DXERR_INVALIDDATA, E_OUTOFMEMORY.

Remarks

Every call to Begin must be matched with a call to **ID3DX10Sprite::End**.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Sprite

D3DX Interfaces

ID3DX10Sprite::DrawSpritesBuffered method

Article • 03/15/2021 • 2 minutes to read

Add an array of sprites to the batch of sprites to be rendered. This must be called in between calls to ID3DX10Sprite::Begin and ID3DX10Sprite::End, and ID3DX10Sprite::Flush must be called before End to send all of the batched sprites to the device for rendering. This draw method is most useful when drawing a small number of sprites that you want buffered into a large batch, such as fonts.

Syntax

C++

```
HRESULT DrawSpritesBuffered(
  [in] D3DX10_SPRITE *pSprites,
  [in] UINT          cSprites
);
```

Parameters

pSprites [in]

Type: D3DX10_SPRITE*

The array of sprites to draw. See D3DX10_SPRITE.

cSprites [in]

Type: UINT

The number of sprites in pSprites.

Return value

Type: HRESULT

If the method succeeds, the return value is S_OK. If the method fails, the return value can be one of the following: D3DERR_INVALIDCALL, D3DXERR_INVALIDDATA.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Sprite

D3DX Interfaces

ID3DX10Sprite::DrawSpritesImmediate method

Article • 03/15/2021 • 2 minutes to read

Draw an array of sprites. This will immediately send the sprites to the device for rendering, which is different from **ID3DX10Sprite::DrawSpritesBuffered** which only adds an array of sprites to a batch of sprites to be rendered when **ID3DX10Sprite::Flush** is called. This draw method is most useful when drawing a large number of sprites that have already been sorted on the CPU (or do not need to be sorted), such as in a particle system. This must be called in between calls to **ID3DX10Sprite::Begin** and **ID3DX10Sprite::End**.

Syntax

C++

```
HRESULT DrawSpritesImmediate(
  [in] D3DX10_SPRITE *pSprites,
  [in] UINT          cSprites,
  [in] UINT          cbSprite,
  [in] UINT          flags
);
```

Parameters

pSprites [in]

Type: **D3DX10_SPRITE***

The array of sprites to draw. See **D3DX10_SPRITE**.

cSprites [in]

Type: **UINT**

The number of sprites in pSprites.

cbSprite [in]

Type: **UINT**

The size of the sprite structure you are passing into pSprites. Passing in 0 is the equivalent of passing in sizeof(D3DX10_SPRITE).

flags [in]

Type: **UINT**

Reserved.

Return value

Type: **HRESULT**

If the method succeeds, the return value is S_OK. If the method fails, the return value can be one of the following: D3DERR_INVALIDCALL, D3DXERR_INVALIDDATA.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Sprite

D3DX Interfaces

ID3DX10Sprite::End method

Article • 03/15/2021 • 2 minutes to read

Call this after ID3DX10Sprite::Flush. If D3DX10_SPRITE_SAVE_STATE was specified when ID3DX10Sprite::Begin was called, this API will restore the device state to how it was before ID3DX10Sprite::Begin was called.

Syntax

```
C++

HRESULT End();
```

Parameters

This method has no parameters.

Return value

Type: **HRESULT**

If the method succeeds, the return value is S_OK. If the method fails, the following value will be returned: D3DERR_INVALIDCALL.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Sprite

D3DX Interfaces

ID3DX10Sprite::Flush method

Article • 03/15/2021 • 2 minutes to read

Force all batched sprites to be submitted to the device. Device states remain as they were after the last call to ID3DX10Sprite::Begin. The list of batched sprites is then cleared.

Syntax

```
C++

HRESULT Flush();
```

Parameters

This method has no parameters.

Return value

Type: **HRESULT**

If the method succeeds, the return value is S_OK. If the method fails, the following value will be returned: D3DERR_INVALIDCALL.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Sprite

D3DX Interfaces

ID3DX10Sprite::GetDevice method

Article • 03/15/2021 • 2 minutes to read

Retrieve the device associated with the sprite object.

Syntax

```
C++

HRESULT GetDevice(
  [out] ID3D10Device **ppDevice
);
```

Parameters

ppDevice [out]

Type: **ID3D10Device****

Address of a pointer to an ID3D10Device interface, representing the Direct3D device object associated with the sprite object.

Return value

Type: **HRESULT**

If the method succeeds, the return value is S_OK. If the method fails, the following value will be returned: D3DERR_INVALIDCALL.

Remarks

Calling this method will increase the internal reference count on the ID3D10Device interface.

Requirements

Requirement	Value
Header	D3DX10.h

Requirement	Value
Library	D3DX10.lib

See also

ID3DX10Sprite

D3DX Interfaces

ID3DX10Sprite::GetProjectionTransform method

Article • 03/15/2021 • 2 minutes to read

Get the sprite projection matrix that is applied to all sprites.

Syntax

```
C++

HRESULT GetProjectionTransform(
  [out] D3DXMATRIX *pProjectionTransform
);
```

Parameters

pProjectionTransform [out]

Type: **D3DXMATRIX***

Pointer to a **D3DX10MATRIX** that will be set to the sprite's projection matrix.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Sprite

ID3DX10Sprite::GetViewTransform method

Article • 03/15/2021 • 2 minutes to read

Get the view transform that applies to all sprites.

Syntax

```
C++

HRESULT GetViewTransform(
  [out] D3DXMATRIX *pViewTransform
);
```

Parameters

pViewTransform [out]

Type: **D3DXMATRIX***

Pointer to a **D3DX10MATRIX** that will be set to the transform of the sprite from the original world space.

Return value

Type: **HRESULT**

If the method succeeds, the return value is S_OK. If the method fails, the following value will be returned: D3DERR_INVALIDCALL.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Sprite

D3DX Interfaces

ID3DX10Sprite::SetProjectionTransform method

Article • 03/15/2021 • 2 minutes to read

Set the projection matrix for all sprites.

Syntax

```
C++

HRESULT SetProjectionTransform(
  [in] D3DXMATRIX *pProjectionTransform
);
```

Parameters

pProjectionTransform [in]

Type: **D3DXMATRIX***

The projection matrix to be used on all sprites.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Sprite

ID3DX10Sprite::SetViewTransform method

Article • 03/15/2021 • 2 minutes to read

Set the view transform that applies to all sprites.

Syntax

C++

```
HRESULT SetViewTransform(
  [in] D3DXMATRIX *pViewTransform
);
```

Parameters

pViewTransform [in]

Type: **D3DXMATRIX***

Pointer to a D3DXMATRIX that contains a transform of the sprite from the original world space. Use this transform to scale, rotate, or transform the sprite.

Return value

Type: **HRESULT**

If the method succeeds, the return value is S_OK. If the method fails, the following value will be returned: D3DERR_INVALIDCALL.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10Sprite

D3DX Interfaces

ID3DX10ThreadPump interface

Article • 05/24/2021 • 4 minutes to read

Used to execute tasks asynchronously, and created with D3DX10CreateThreadPump. There are several D3DX10 APIs that can optionally take a thread pump as a parameter, such as D3DX10CreateTextureFromFile and D3DX10CompileFromFile (see remarks for complete list). If the thread pump is passed into these APIs, they will execute asynchronously on a separate thread pump thread. The advantage to doing this is that it can make loading and processing large amounts of data happen without seeing a noticable slow down in performance on screen.

Members

The **ID3DX10ThreadPump** interface inherits from the IUnknown interface. **ID3DX10ThreadPump** also has these types of members:

- Methods

Methods

The **ID3DX10ThreadPump** interface has these methods.

Method	Description
AddWorkItem	Add a work item to the thread pump.
GetQueueStatus	Get the number of items in each of the three queues inside the thread pump.
GetWorkItemCount	Get the number of work items currently in the thread pump.
ProcessDeviceWorkItems	Set work items to the device after they have finished loading and processing. 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. See remarks.
PurgeAllItems	Clear all work items from the thread pump.
WaitForAllItems	Wait for all work items in the thread pump to finish.

Remarks

The thread pump loads and processes data in a 3 step process. It goes:

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: **ID3DX10DataLoader::Load**, **ID3DX10DataLoader::Decompress**, and **ID3DX10DataLoader::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: **ID3DX10DataProcessor::Process**, **ID3DX10DataProcessor::CreateDeviceObject**, and **ID3DX10DataProcessor::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 **ID3DX10DataProcessor::Process** will do JPEG decompression to get the image's raw image bits. If the data is a shader, then **ID3DX10DataProcessor::Process** will compile the HLSL into bytecode. After the data has been processed a device object will be created for that data (with **ID3DX10DataProcessor::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 **ID3DX10ThreadPump::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 **D3DX10CreateTextureFromFile** and **D3DX10CompileFromFile**, or by calling **ID3DX10ThreadPump::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. D3DX10 provides a set of APIs for creating data loaders and data processors that have functionality for loading and processing common data formats (see remarks for complete list of APIs). 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.

Built-in D3DX10 data loaders

	Description
D3DX10CreateAsyncFileLoader	Create a file loader asynchronously.
D3DX10CreateAsyncMemoryLoader	Create a data loader asynchronously.
D3DX10CreateAsyncResourceLoader	Create a resource loader asynchronously.

Built-in D3DX10 data processors

	Description
D3DX10CreateAsyncTextureProcessor	Create a data processor to be used with a thread pump. This API is similar to D3DX10CreateAsyncTextureInfoProcessor but it also loads the texture.
D3DX10CreateAsyncTextureInfoProcessor	Create a data processor to be used with a thread pump.
D3DX10CreateAsyncShaderCompilerProcessor	Compile a shader and create a data processor asynchronously.
D3DX10CreateAsyncEffectCompilerProcessor	Create an effect with a data processor asynchronously.
D3DX10CreateAsyncEffectCreateProcessor	Create an effect pool asynchronously.
D3DX10CreateAsyncEffectPoolCreateProcessor	Create a data processor asynchronously.
D3DX10CreateAsyncShaderPreprocessProcessor	Create a data processor for a shader asynchronously.

APIs that take a thread pump as a parameter.

	Description
D3DX10CompileFromFile	Compile a shader from a file.
D3DX10CompileFromMemory	Compile a shader residing in memory.
D3DX10CompileFromResource	Compile a shader from a resource.
D3DX10CreateEffectFromFile	Create an effect from a file.

	Description
D3DX10CreateEffectFromMemory	Create an effect from memory.
D3DX10CreateEffectFromResource	Create an effect from a resource.
D3DX10CreateEffectPoolFromFile	Create an effect pool from a file.
D3DX10CreateEffectPoolFromMemory	Create an effect pool from a file residing in memory.
D3DX10CreateEffectPoolFromResource	Create an effect pool from a resource.
D3DX10PreprocessShaderFromFile	Create a shader from a file without compiling it.
D3DX10PreprocessShaderFromMemory	Create a shader from memory without compiling it.
D3DX10PreprocessShaderFromResource	Create a shader from a resource without compiling it.
D3DX10CreateShaderResourceViewFromFile	Create a shader-resource view from a file.
D3DX10CreateShaderResourceViewFromMemory	Create a shader-resource view from a file in memory.
D3DX10CreateShaderResourceViewFromResource	Create a shader-resource view from a resource.
D3DX10GetImageInfoFromFile	Retrieves information about a given image file.
D3DX10GetImageInfoFromMemory	Get information about an image already loaded into memory.
D3DX10GetImageInfoFromResource	Retrieves information about a given image in a resource.
D3DX10CreateTextureFromFile	Create a texture resource from a file.
D3DX10CreateTextureFromMemory	Create a texture resource from a file residing in system memory.
D3DX10CreateTextureFromResource	Create a texture resource from another resource.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

D3DX Interfaces

ID3DX10ThreadPump::AddWorkItem method

Article • 03/15/2021 • 2 minutes to read

Add a work item to the thread pump.

Syntax

```
C++

HRESULT AddWorkItem(
  [in]  ID3DX10DataLoader    *pDataLoader,
  [in]  ID3DX10DataProcessor *pDataProcessor,
  [in]  HRESULT              *pHResult,
  [out] void                 **ppDeviceObject
);
```

Parameters

pDataLoader [in]

Type: **ID3DX10DataLoader***

The loader that the thread pump will use when a work item requires data to be loaded.

pDataProcessor [in]

Type: **ID3DX10DataProcessor***

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 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10ThreadPump

D3DX Interfaces

ID3DX10ThreadPump::GetQueueStatus method

Article • 03/15/2021 • 2 minutes to read

Get the number of items in each of the three queues inside the thread pump.

Syntax

```
C++

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 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10ThreadPump

D3DX Interfaces

ID3DX10ThreadPump::GetWorkItemCount method

Article • 03/15/2021 • 2 minutes to read

Get the number of work items currently in the thread pump.

Syntax

```
C++

UINT GetWorkItemCount();
```

Parameters

This method has no parameters.

Return value

Type: **UINT**

The number of work items currently queued in the thread pump.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10ThreadPump

D3DX Interfaces

ID3DX10ThreadPump::ProcessDeviceWorkItems method

Article • 03/15/2021 • 2 minutes to read

Set work items to the device after they have finished loading and processing. 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. See remarks.

Syntax

```
C++

HRESULT ProcessDeviceWorkItems(
  [in] UINT iWorkItemCount
);
```

Parameters

iWorkItemCount [in]

Type: **UINT**

The number of work items to set to the device. ProcessDeviceObjectCreation will create at most iWorkItemCount objects. If there are not enough work items in the queue to process iWorkItemCount objects, ProcessDeviceObjectCreation will create as many device objects as there are items in the queue.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Remarks

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 noticable 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 and minimizing the possibility of a noticable slow down in the game's performance.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10ThreadPump

D3DX Interfaces

ID3DX10ThreadPump::PurgeAllItems method

Article • 03/15/2021 • 2 minutes to read

Clear all work items from the thread pump.

Syntax

```
C++

HRESULT PurgeAllItems();
```

Parameters

This method has no parameters.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10ThreadPump

D3DX Interfaces

ID3DX10ThreadPump::WaitForAllItems method

Article • 03/15/2021 • 2 minutes to read

Wait for all work items in the thread pump to finish.

Syntax

```
C++

HRESULT WaitForAllItems();
```

Parameters

This method has no parameters.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DX10ThreadPump

D3DX Interfaces

ID3DXMatrixStack interface

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead along with this header from **GitHub**.

Applications use the methods of the ID3DXMATRIXStack interface to manipulate a matrix stack.

Members

The **ID3DXMatrixStack** interface inherits from the **IUnknown** interface. **ID3DXMatrixStack** also has these types of members:

- Methods

Methods

The **ID3DXMatrixStack** interface has these methods.

Method	Description
GetTop	Retrieves the current matrix at the top of the stack.
LoadIdentity	Loads identity in the current matrix.
LoadMatrix	Loads the given matrix into the current matrix.
MultMatrix	Determines the product of the current matrix and the given matrix.
MultMatrixLocal	Determines the product of the given matrix and the current matrix.
Pop	Removes the current matrix from the top of the stack.
Push	Adds a matrix to the stack.
RotateAxis	Rotates (relative to world coordinate space) around an arbitrary axis.
RotateAxisLocal	Rotates (relative to the object's local coordinate space) around an arbitrary axis.
RotateYawPitchRoll	Rotates (relative to world coordinate space) around an arbitrary axis.

Method	Description
RotateYawPitchRollLocal	Rotates (relative to the object's local coordinate space) around an arbitrary axis.
Scale	Scale the current matrix about the world coordinate origin.
ScaleLocal	Scale the current matrix about the object origin.
Translate	Determines the product of the current matrix and the computed translation matrix determined by the given factors (x, y, and z).
TranslateLocal	Determines the product of the computed translation matrix determined by the given factors (x, y, and z) and the current matrix.

Remarks

The ID3DX10MATRIXStack interface is obtained by calling the D3DXCreateMatrixStack function.

The LPD3DX10MATRIXSTACK type is defined as a pointer to the **ID3DXMatrixStack** interface.

```
typedef interface ID3DXMatrixStack ID3DXMatrixStack;
typedef interface ID3DXMatrixStack *LPD3DXMATRIXSTACK;
```

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

D3DX Interfaces

ID3DXMATRIXStack::GetTop method (D3DX10.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead along with this header from **GitHub**.

Retrieves the current matrix at the top of the stack.

Syntax

```C++
D3DXMATRIX* GetTop();
```

Parameters

This method has no parameters.

Return value

Type: **D3DXMATRIX***

This method returns a pointer to a D3DXMATRIX structure representing the current matrix.

Remarks

The D3DXMATRIX pointer returned by this method is not guaranteed to be valid after subsequent stack operations.

Note that this method does not remove the current matrix from the top of the stack; rather, it just returns the current matrix.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DXMatrixStack

D3DX Interfaces

ID3DXMATRIXStack::LoadIdentity method (D3DX10.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead along with this header from **GitHub**.

Loads identity in the current matrix.

Syntax

```
C++

HRESULT LoadIdentity();
```

Parameters

This method has no parameters.

Return value

Type: **HRESULT**

If the method succeeds, the return value is D3D_OK. If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

The identity matrix is a matrix in which all coefficients are 0.0 except the [1,1][2,2][3,3][4,4] coefficients, which are set to 1.0. The identity matrix is special in that when it is applied to vertices, they are unchanged. The identity matrix is used as the starting point for matrices that will modify vertex values to create rotations, translations, and any other transformations that can be represented by a 4x4 matrix.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DXMatrixStack

D3DX Interfaces

ID3DXMATRIXStack::LoadMatrix method (D3DX10.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead along with this header from **GitHub**.

Loads the given matrix into the current matrix.

Syntax

```
C++

HRESULT LoadMatrix(
  [in] const D3DXMATRIX *pM
);
```

Parameters

pM [in]

Type: **const D3DXMATRIX***

Pointer to the D3DXMATRIX structure loaded into the current matrix.

Return value

Type: **HRESULT**

If the method succeeds, the return value is D3D_OK. If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

Note that this method does not add an item to the stack; rather, it replaces the current matrix with the supplied matrix.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DXMatrixStack

D3DX Interfaces

ID3DXMATRIXStack::MultMatrix method (D3DX10.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead along with this header from **GitHub**.

Determines the product of the current matrix and the given matrix.

Syntax

```C++
HRESULT MultMatrix(
  [in] const D3DXMATRIX *pM
);
```

Parameters

pM [in]

Type: **const D3DXMATRIX***

Pointer to the D3DXMATRIX structure to be multiplied with the current matrix.

Return value

Type: **HRESULT**

If the method succeeds, the return value is D3D_OK. If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

This method right-multiplies the given matrix to the current matrix (transformation is about the current world origin).

```
m_pstack[m_currentPos] = m_pstack[m_currentPos] * (*pMat);
```

This method does not add an item to the stack, it replaces the current matrix with the product of the current matrix and the given matrix.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DXMatrixStack

D3DX Interfaces

ID3DXMATRIXStack::MultMatrixLocal method (D3DX10.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead along with this header from **GitHub**.

Determines the product of the given matrix and the current matrix.

Syntax

```C++
HRESULT MultMatrixLocal(
  [in] const D3DXMATRIX *pM
);
```

Parameters

pM [in]

Type: **const D3DXMATRIX***

Pointer to the D3DXMATRIX structure to be multiplied with the current matrix.

Return value

Type: **HRESULT**

If the method succeeds, the return value is D3D_OK. If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

This method left-multiplies the given matrix to the current matrix (transformation is about the local origin of the object).

```
m_pstack[m_currentPos] = (*pMat) * m_pstack[m_currentPos];
```

This method does not add an item to the stack, it replaces the current matrix with the product of the given matrix and the current matrix.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DXMatrixStack

D3DX Interfaces

ID3DXMATRIXStack::Pop method (D3DX10.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead along with this header from **GitHub**.

Removes the current matrix from the top of the stack.

Syntax

```C++
HRESULT Pop();
```

Parameters

This method has no parameters.

Return value

Type: **HRESULT**

If the method succeeds, the return value is D3D_OK.

Remarks

Note that this method decrements the count of items on the stack by 1, effectively removing the current matrix from the top of the stack and promoting a matrix to the top of the stack. If the current count of items on the stack is 0, then this method returns without performing any action. If the current count of items on the stack is 1, then this method empties the stack.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DXMatrixStack

D3DX Interfaces

ID3DXMATRIXStack::Push method (D3DX10.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead along with this header from **GitHub**.

Adds a matrix to the stack.

Syntax

```
C++

HRESULT Push();
```

Parameters

This method has no parameters.

Return value

Type: **HRESULT**

If the method succeeds, the return value is D3D_OK. If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

This method increments the count of items on the stack by 1, duplicating the current matrix. The stack will grow dynamically as more items are added.

Requirements

Requirement	Value

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DXMatrixStack

D3DX Interfaces

ID3DXMATRIXStack::RotateAxis method (D3DX10.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead along with this header from **GitHub**.

Rotates (relative to world coordinate space) around an arbitrary axis.

Syntax

```C++
HRESULT RotateAxis(
  [in] const D3DXVECTOR3 *pV,
  [in]       FLOAT       Angle
);
```

Parameters

pV [in]

Type: **const** D3DXVECTOR3*

Pointer to the arbitrary axis of rotation. See D3DXVECTOR3.

Angle [in]

Type: FLOAT

Rotation angle about the arbitrary axis, in radians. Angles are measured counterclockwise when looking along the arbitrary axis toward the origin.

Return value

Type: HRESULT

If the method succeeds, the return value is D3D_OK. If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

This method adds the rotation to the matrix stack with the computed rotation matrix similar to the following:

```
D3DXMATRIX tmp;
D3DXMatrixRotationAxis( &tmp, pV, angle );
m_stack[m_currentPos] = m_stack[m_currentPos] * tmp;
```

Because the rotation is right-multiplied to the matrix stack, the rotation is relative to world coordinate space.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DXMatrixStack

D3DX Interfaces

ID3DXMATRIXStack::RotateAxisLocal method (D3DX10.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead along with this header from **GitHub**.

Rotates (relative to the object's local coordinate space) around an arbitrary axis.

Syntax

```C++
HRESULT RotateAxisLocal(
  [in] const D3DXVECTOR3 *pV,
  [in]       FLOAT       Angle
);
```

Parameters

pV [in]

Type: **const** D3DXVECTOR3*

Pointer to the arbitrary axis of rotation. See D3DXVECTOR3.

Angle [in]

Type: FLOAT

Rotation angle about the arbitrary axis, in radians. Angles are measured counterclockwise when looking along the arbitrary axis toward the origin.

Return value

Type: HRESULT

If the method succeeds, the return value is D3D_OK. If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

This method adds the rotation to the matrix stack with the computed rotation matrix similar to the following:

```
D3DXMATRIX tmp;
D3DXMatrixRotationAxis( &tmp, pV, angle );
m_stack[m_currentPos] = tmp * m_stack[m_currentPos];
```

Because the rotation is left-multiplied to the matrix stack, the rotation is relative to the object's local coordinate space.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DXMatrixStack

D3DX Interfaces

ID3DXMATRIXStack::RotateYawPitchRoll method (D3DX10.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead along with this header from **GitHub**.

Rotates (relative to world coordinate space) around an arbitrary axis.

Syntax

```C++
HRESULT RotateYawPitchRoll(
  [in] FLOAT Yaw,
  [in] FLOAT Pitch,
  [in] FLOAT Roll
);
```

Parameters

Yaw [in]

Type: **FLOAT**

The yaw around the y-axis in radians.

Pitch [in]

Type: **FLOAT**

The pitch around the x-axis in radians.

Roll [in]

Type: **FLOAT**

The roll around the z-axis in radians.

Return value

Type: HRESULT

If the method succeeds, the return value is D3D_OK.

Remarks

This method adds the rotation to the matrix stack with the computed rotation matrix similar to the following:

```
D3DXMATRIX tmp;
D3DXMatrixRotationYawPitchRoll( &tmp, yaw, pitch, roll );
m_stack[m_currentPos] = m_stack[m_currentPos] * tmp;
```

Because the rotation is right-multiplied to the matrix stack, the rotation is relative to world coordinate space.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DXMatrixStack

D3DX Interfaces

ID3DXMATRIXStack::RotateYawPitchRollLocal method (D3DX10.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead along with this header from **GitHub**.

Rotates (relative to the object's local coordinate space) around an arbitrary axis.

Syntax

```C++
HRESULT RotateYawPitchRollLocal(
  [in] FLOAT Yaw,
  [in] FLOAT Pitch,
  [in] FLOAT Roll
);
```

Parameters

Yaw [in]

Type: **FLOAT**

The yaw around the y-axis in radians.

Pitch [in]

Type: **FLOAT**

The pitch around the x-axis in radians.

Roll [in]

Type: **FLOAT**

The roll around the z-axis in radians.

Return value

Type: HRESULT

If the method succeeds, the return value is D3D_OK.

Remarks

This method adds the rotation to the matrix stack with the computed rotation matrix similar to the following:

```
D3DXMATRIX tmp;
D3DXMatrixRotationYawPitchRoll( &tmp, yaw, pitch, roll );
m_stack[m_currentPos] = tmp * m_stack[m_currentPos];
```

Because the rotation is left-multiplied to the matrix stack, the rotation is relative to the object's local coordinate space.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DXMatrixStack

D3DX Interfaces

ID3DXMATRIXStack::Scale method (D3DX10.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead along with this header from **GitHub**.

Scale the current matrix about the world coordinate origin.

Syntax

```C++
HRESULT Scale(
  [in] FLOAT x,
  [in] FLOAT y,
  [in] FLOAT z
);
```

Parameters

x [in]

Type: **FLOAT**

The scaling component in the x-direction.

y [in]

Type: **FLOAT**

The scaling component in the y-direction.

z [in]

Type: **FLOAT**

The scaling component in the z-direction.

Return value

Type: HRESULT

If the method succeeds, the return value is D3D_OK.

Remarks

This method right-multiplies the current matrix with the computed scale matrix. The transformation is about the current world origin.

```
D3DXMATRIX tmp;
D3DXMatrixScaling(&tmp, x, y, z);
m_stack[m_currentPos] = m_stack[m_currentPos] * tmp;
```

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DXMatrixStack

D3DX Interfaces

ID3DXMATRIXStack::ScaleLocal method (D3DX10.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead along with this header from **GitHub**.

Scale the current matrix about the object origin.

Syntax

```C++
HRESULT ScaleLocal(
  [in] FLOAT x,
  [in] FLOAT y,
  [in] FLOAT z
);
```

Parameters

x [in]

Type: **FLOAT**

The scaling component in the x-direction.

y [in]

Type: **FLOAT**

The scaling component in the y-direction.

z [in]

Type: **FLOAT**

The scaling component in the z-direction.

Return value

Type: HRESULT

If the method succeeds, the return value is D3D_OK.

Remarks

This method left-multiplies the current matrix with the computed scale matrix. The transformation is about the local origin of the object.

```
D3DXMATRIX tmp;
D3DXMatrixScaling(&tmp, x, y, z);
m_stack[m_currentPos] = tmp * m_stack[m_currentPos];
```

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DXMatrixStack

D3DX Interfaces

ID3DXMATRIXStack::Translate method (D3DX10.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead along with this header from **GitHub**.

Determines the product of the current matrix and the computed translation matrix determined by the given factors (x, y, and z).

Syntax

```C++
HRESULT Translate(
  [in] FLOAT x,
  [in] FLOAT y,
  [in] FLOAT z
);
```

Parameters

x [in]

Type: **FLOAT**

The translation factor in the x-direction.

y [in]

Type: **FLOAT**

The translation factor in the y-direction.

z [in]

Type: **FLOAT**

The translation factor in the z-direction.

Return value

Type: HRESULT

If the method succeeds, the return value is D3D_OK.

Remarks

This method right-multiplies the current matrix with the computed translation matrix (transformation is about the current world origin).

```
D3DXMATRIX tmp;
D3DXMatrixTranslation( &tmp, x, y, z );
m_stack[m_currentPos] = m_stack[m_currentPos] * tmp;
```

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DXMatrixStack

D3DX Interfaces

ID3DXMATRIXStack::TranslateLocal method (D3DX10.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead along with this header from **GitHub**.

Determines the product of the computed translation matrix determined by the given factors (x, y, and z) and the current matrix.

Syntax

```C++
HRESULT TranslateLocal(
  [in] FLOAT x,
  [in] FLOAT y,
  [in] FLOAT z
);
```

Parameters

x [in]

Type: **FLOAT**

The translation factor in the x-direction.

y [in]

Type: **FLOAT**

The translation factor in the y-direction.

z [in]

Type: **FLOAT**

The translation factor in the z-direction.

Return value

Type: HRESULT

If the method succeeds, the return value is D3D_OK.

Remarks

This method left-multiplies the current matrix with the computed translation matrix (transformation is about the local origin of the object).

```
D3DXMATRIX tmp;
D3DXMatrixTranslation(&tmp, x, y, z );
m_stack[m_currentPos] = tmp * m_stack[m_currentPos];
```

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

ID3DXMatrixStack

D3DX Interfaces

D3DX Functions (Direct3D 10 Graphics)

Article • 01/06/2021 • 2 minutes to read

This section contains reference information for the functions that you need to implement when you work with the D3DX utility library. The following categories of functions are implemented:

- General Purpose Functions
- Math Functions
- Mesh Functions
- Texture Functions in D3DX 10

Related topics

D3DX Reference

General Purpose Functions (Direct3D 10 Graphics)

Article • 01/06/2021 • 2 minutes to read

D3DX supplies the following general purpose functions.

- **D3DX10CheckVersion**
- D3DX10CompileEffectFromFile
- D3DX10CompileEffectFromMemory
- D3DX10CompileEffectFromResource
- **D3DX10CompileFromFile**
- **D3DX10CompileFromMemory**
- **D3DX10CompileFromResource**
- D3DX10CompileShaderFromFile
- D3DX10CompileShaderFromMemory
- D3DX10CompileShaderFromResource
- **D3DX10CreateAsyncCompilerProcessor**
- **D3DX10CreateAsyncEffectCompilerProcessor**
- **D3DX10CreateAsyncEffectCreateProcessor**
- **D3DX10CreateAsyncEffectPoolCreateProcessor**
- **D3DX10CreateAsyncFileLoader**
- **D3DX10CreateAsyncMemoryLoader**
- **D3DX10CreateAsyncResourceLoader**
- **D3DX10CreateAsyncShaderCompilerProcessor**
- **D3DX10CreateAsyncShaderPreprocessProcessor**
- **D3DX10CreateAsyncShaderResourceViewProcessor**
- **D3DX10CreateDevice**
- **D3DX10CreateDeviceAndSwapChain**
- **D3DX10CreateEffectFromFile**
- **D3DX10CreateEffectFromMemory**
- **D3DX10CreateEffectFromResource**
- **D3DX10CreateEffectPoolFromFile**
- **D3DX10CreateEffectPoolFromMemory**
- **D3DX10CreateEffectPoolFromResource**
- **D3DX10CreateFont**
- **D3DX10CreateFontIndirect**
- **D3DX10CreateSprite**
- **D3DX10CreateThreadPump**
- **D3DX10DisassembleEffect**

- **D3DX10DisassembleShader**
- **D3DX10DebugMute**
- D3DX10GetDriverLevel
- **D3DX10GetFeatureLevel1**
- **D3DX10PreprocessShaderFromFile**
- **D3DX10PreprocessShaderFromMemory**
- **D3DX10PreprocessShaderFromResource**
- **D3DX10ReflectShader**
- **D3DX10UnsetAllDeviceObjects**

Related topics

D3DX Functions

General Purpose Functions

D3DX10CheckVersion function

Article • 04/27/2021 • 2 minutes to read

Verify that the version of D3DX you compiled with is the version that you are running.

Syntax

```
C++

HRESULT D3DX10CheckVersion(
  _In_ UINT D3DSdkVersion,
  _In_ UINT D3DX10SdkVersion
);
```

Parameters

D3DSdkVersion [in]

Type: **UINT**

Use D3D10_SDK_VERSION. See remarks.

D3DX10SdkVersion [in]

Type: **UINT**

Use D3DX10_SDK_VERSION. See remarks.

Return value

Type: **HRESULT**

If the version doesn't match, the function will return FALSE (a number less than or equal to 0, the number itself has no meaning).

Remarks

Use this function during the initialization of your application.

```
HRESULT hr;

if( FAILED( D3DX10CheckVersion(D3D10_SDK_VERSION, D3DX10_SDK_VERSION) ) )
    return E_FAIL;
```

Requirements

Requirement	Value
Header	D3DX10Core.h
Library	D3DX10.lib

See also

General Purpose Functions

D3DX10CompileEffectFromFile

Article • 06/18/2021 • 2 minutes to read

> **Note**
>
> Instead of using this legacy function, we recommend that you compile offline by using the Fxc.exe command-line compiler or use the **D3DCompile** API.

Deprecated. See **D3DX10CompileFromFile**.

Related topics

General Purpose Functions

D3DX10CompileEffectFromMemory

Article • 06/18/2021 • 2 minutes to read

> **Note**
>
> Instead of using this legacy function, we recommend that you compile offline by using the Fxc.exe command-line compiler or use the **D3DCompile** API.

Deprecated. See **D3DX10CompileFromMemory**.

Related topics

General Purpose Functions

D3DX10CompileEffectFromResource

Article • 06/18/2021 • 2 minutes to read

> **Note**
>
> Instead of using this legacy function, we recommend that you compile offline by using the Fxc.exe command-line compiler or use the **D3DCompile** API.

Deprecated. See **D3DX10CompileFromResource**.

Related topics

General Purpose Functions

D3DX10CompileShaderFromFile

Article • 06/18/2021 • 2 minutes to read

> **Note**
>
> Instead of using this legacy function, we recommend that you compile offline by using the Fxc.exe command-line compiler or use the **D3DCompile** API.

Deprecated. See **D3DX10CompileFromFile**.

Related topics

General Purpose Functions

D3DX10CompileShaderFromMemory

Article • 06/18/2021 • 2 minutes to read

> **Note**
>
> Instead of using this legacy function, we recommend that you compile offline by using the Fxc.exe command-line compiler or use the **D3DCompile** API.

Deprecated. See **D3DX10CompileFromMemory**.

Related topics

General Purpose Functions

D3DX10CompileShaderFromResource

Article • 06/18/2021 • 2 minutes to read

> **Note**
>
> Instead of using this legacy function, we recommend that you compile offline by using the Fxc.exe command-line compiler or use the **D3DCompile** API.

Deprecated. See **D3DX10CompileFromResource**.

Related topics

General Purpose Functions

D3DX10CompileFromFile function

Article • 03/15/2021 • 2 minutes to read

> **Note**
>
> Instead of using this legacy function, we recommend that you compile offline by using the Fxc.exe command-line compiler or use the **D3DCompile** API.

Compile a shader or an effect from a file.

Syntax

C++

```
HRESULT D3DX10CompileFromFile(
  _In_        LPCTSTR            pSrcFile,
  _In_  const D3D_SHADER_MACRO *pDefines,
  _In_        LPD3D10INCLUDE     pInclude,
  _In_        LPCSTR             pFunctionName,
  _In_        LPCSTR             pProfile,
  _In_        UINT               Flags1,
  _In_        UINT               Flags2,
  _In_        ID3DX10ThreadPump  *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 D3D_SHADER_MACRO***

Optional. Pointer to an array of macro definitions (see **D3D_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 **ID3D10Include Interface** 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, **D3DX10CompileFromFile** 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, or shader model 4.

Flags1 [in]

Type: **UINT**

Shader compile flags.

Flags2 [in]

Type: **UINT**

Effect compile flags. When you compile a shader and not an effect file, **D3DX10CompileFromFile** 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: **ID3DX10ThreadPump***

A pointer to a thread pump interface (see **ID3DX10ThreadPump Interface**). Use **NULL** to specify that this function should not return until it is completed.

ppShader [out]

Type: **ID3D10Blob****

A pointer to an **ID3D10Blob Interface** which contains the compiled shader, as well as any embedded debug and symbol-table information.

ppErrorMsgs [out]

Type: **ID3D10Blob****

A pointer to an **ID3D10Blob Interface** 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 10 Return Codes.

Remarks

Requirements

Requirement	Value
Header	D3DX10Async.h

See also

General Purpose Functions

D3DX10CompileFromMemory function

Article • 03/15/2021 • 2 minutes to read

> **Note**
>
> Instead of using this legacy function, we recommend that you compile offline by using the Fxc.exe command-line compiler or use the **D3DCompile** API.

Compile a shader or an effect that is loaded in memory.

Syntax

C++

```
HRESULT D3DX10CompileFromMemory(
  _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_        ID3DX10ThreadPump  *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 D3D_SHADER_MACRO***

Optional. Pointer to an array of macro definitions (see **D3D_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 **ID3D10Include Interface** 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, **D3DX10CompileFromMemory** 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, or shader model 4.

Flags1 [in]

Type: **UINT**

Shader compile flags.

Flags2 [in]

Type: **UINT**

Effect compile flags. When you compile a shader and not an effect file, **D3DX10CompileFromMemory** 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: **ID3DX10ThreadPump***

A pointer to a thread pump interface (see **ID3DX10ThreadPump Interface**). Use **NULL** to specify that this function should not return until it is completed.

ppShader [out]

Type: **ID3D10Blob****

A pointer to an **ID3D10Blob Interface** which contains the compiled shader, as well as any embedded debug and symbol-table information.

ppErrorMsgs [out]

Type: **ID3D10Blob****

A pointer to an **ID3D10Blob Interface** 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 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10Async.h

See also

General Purpose Functions

D3DX10CompileFromResource function

Article • 03/15/2021 • 2 minutes to read

> **Note**
>
> Instead of using this legacy function, we recommend that you compile offline by using the Fxc.exe command-line compiler or use the **D3DCompile** API.

Compile a shader or an effect from a resource.

Syntax

C++

```
HRESULT D3DX10CompileFromResource(
  _In_        HMODULE            hSrcModule,
  _In_        LPCTSTR            pSrcResource,
  _In_        LPCTSTR            pSrcFileName,
  _In_  const D3D_SHADER_MACRO *pDefines,
  _In_        LPD3D10INCLUDE     pInclude,
  _In_        LPCSTR             pFunctionName,
  _In_        LPCSTR             pProfile,
  _In_        UINT               Flags1,
  _In_        UINT               Flags2,
  _In_        ID3DX10ThreadPump  *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 D3D_SHADER_MACRO***

Optional. Pointer to an array of macro definitions (see **D3D_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 **ID3D10Include Interface** 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, **D3DX10CompileFromResource** 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, or shader model 4.

Flags1 [in]

Type: **UINT**

Shader compile flags.

Flags2 [in]

Type: **UINT**

Effect compile flags. When you compile a shader and not an effect file, **D3DX10CompileFromResource** 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: **ID3DX10ThreadPump***

A pointer to a thread pump interface (see **ID3DX10ThreadPump Interface**). Use **NULL** to specify that this function should not return until it is completed.

ppShader [out]

Type: **ID3D10Blob****

A pointer to an **ID3D10Blob Interface** which contains the compiled shader, as well as any embedded debug and symbol-table information.

ppErrorMsgs [out]

Type: **ID3D10Blob****

A pointer to an **ID3D10Blob Interface** 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 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10Async.h

See also

General Purpose Functions

D3DX10CreateAsyncCompilerProcessor function

Article • 03/15/2021 • 2 minutes to read

Create an asynchronous-data processor for a shader.

Syntax

C++

```
HRESULT D3DX10CreateAsyncCompilerProcessor(
  _In_        LPCSTR                pFileName,
  _In_  const D3D10_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_       ID3DX10DataProcessor **ppDataProcessor
);
```

Parameters

pFileName [in]

Type: LPCSTR

A string that contains the shader filename.

pDefines [in]

Type: **const** D3D_SHADER_MACRO*

A NULL-terminated array of shader macros (see D3D_SHADER_MACRO); set this to **NULL** to specify no macros.

pInclude [in]

Type: LPD3D10INCLUDE

A pointer to an include interface (see ID3D10Include 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, **D3DX10CreateAsyncCompilerProcessor** 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.

Flags1 [in]

Type: **UINT**

Shader compile flags.

Flags2 [in]

Type: **UINT**

Effect compile flags. When you compile a shader and not an effect file, **D3DX10CreateAsyncCompilerProcessor** ignores *Flags2*; we recommend that you set *Flags2* to zero because it is good programming practice to set a pointer parameter to **NULL** if the called function will not use it.

ppCompiledShader [out]

Type: **ID3D10Blob****

Address of a pointer to the compiled effect (see **ID3D10Blob Interface**).

ppErrorBuffer [out]

Type: **ID3D10Blob****

Address of a pointer to compile errors (see **ID3D10Blob Interface**).

ppDataProcessor [out]

Type: **ID3DX10DataProcessor****

Address of a pointer to a buffer that contains the data processor created (see **ID3DX10DataProcessor Interface**).

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10Async.h

See also

General Purpose Functions

D3DX10CreateAsyncEffectCompilerProcessor function

Article • 03/15/2021 • 2 minutes to read

Create an asynchronous-data processor for an effect.

Syntax

C++

```
HRESULT D3DX10CreateAsyncEffectCompilerProcessor(
  _In_        LPCSTR                pFileName,
  _In_  const D3D_SHADER_MACRO   *pDefines,
  _In_        LPD3D10INCLUDE        pInclude,
  _In_        UINT                  Flags,
  _In_        UINT                  FXFlags,
  _Out_       ID3D10Blob            **ppCompiledShader,
  _Out_       ID3D10Blob            **ppErrorBuffer,
  _Out_       ID3DX10DataProcessor **ppDataProcessor
);
```

Parameters

pFileName [in]

Type: LPCSTR

A string that contains the effect filename.

pDefines [in]

Type: **const** D3D_SHADER_MACRO*

A NULL-terminated array of shader macros (see D3D_SHADER_MACRO); set this to **NULL** to specify no macros.

pInclude [in]

Type: LPD3D10INCLUDE

A pointer to an include interface (see ID3D10Include Interface). This parameter can be **NULL**.

Flags [in]

Type: **UINT**

HLSL compile options.

FXFlags [in]

Type: **UINT**

Effect compile options).

ppCompiledShader [out]

Type: **ID3D10Blob****

Address of a pointer to buffer (see **ID3D10Blob Interface**) that contains the compiled effect.

ppErrorBuffer [out]

Type: **ID3D10Blob****

Address of a pointer to a buffer (see **ID3D10Blob Interface**) that contains compile errors.

ppDataProcessor [out]

Type: **ID3DX10DataProcessor****

Address of a pointer to a buffer that contains the data processor created (see **ID3DX10DataProcessor Interface**).

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10Async.h

See also

General Purpose Functions

D3DX10CreateAsyncEffectCreateProcessor function

Article • 03/15/2021 • 2 minutes to read

Create an effect pool asynchronously.

Syntax

C++

```
HRESULT D3DX10CreateAsyncEffectCreateProcessor(
  _In_        LPCSTR                pFileName,
  _In_  const D3D_SHADER_MACRO    *pDefines,
  _In_        LPD3D10INCLUDE        pInclude,
  _In_        LPCSTR                pProfile,
  _In_        UINT                  Flags,
  _In_        UINT                  FXFlags,
  _In_        ID3D10Device          *pDevice,
  _In_        ID3D10EffectPool      *pPool,
  _Out_       ID3D10Blob            **ppErrorBuffer,
  _Out_       ID3DX10DataProcessor **ppProcessor
);
```

Parameters

pFileName [in]

Type: LPCSTR

A string that contains the effect filename.

pDefines [in]

Type: **const** D3D_SHADER_MACRO*

A NULL-terminated array of shader macros (see D3D_SHADER_MACRO); set this to **NULL** to specify no macros.

pInclude [in]

Type: LPD3D10INCLUDE

A pointer to an include interface (see ID3D10Include Interface); set this to **NULL** to specify there is no include file.

pProfile [in]

Type: **LPCSTR**

A string that specifies the shader profile or shader model.

Flags [in]

Type: **UINT**

HLSL compile options (see Shader Flags).

FXFlags [in]

Type: **UINT**

Effect compile options (see Compile and Effect Flags).

pDevice [in]

Type: **ID3D10Device***

A pointer to the device (see **ID3D10Device Interface**) that will use the resources.

pPool [in]

Type: **ID3D10EffectPool***

A pointer to an effect pool (see **ID3D10EffectPool Interface**) for sharing variables between effects.

ppErrorBuffer [out]

Type: **ID3D10Blob****

The address of a pointer to memory (see **ID3D10Blob Interface**) that contains effect compile errors, if there were any.

ppProcessor [out]

Type: **ID3DX10DataProcessor****

The address of a pointer to the asynchronous-data processor (see **ID3DX10DataProcessor Interface**).

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10Async.h

See also

General Purpose Functions

D3DX10CreateAsyncEffectPoolCreateProcessor function

Article • 03/15/2021 • 2 minutes to read

Create an asynchronous-data processor for a memory pool.

Syntax

C++

```
HRESULT D3DX10CreateAsyncEffectPoolCreateProcessor(
  _In_        LPCSTR                pFileName,
  _In_  const D3D_SHADER_MACRO   *pDefines,
  _In_        LPD3D10INCLUDE        pInclude,
  _In_        LPCSTR                pProfile,
  _In_        UINT                  Flags,
  _In_        UINT                  FXFlags,
  _In_        ID3D10Device          *pDevice,
  _Out_       ID3D10Blob            **ppErrorBuffer,
  _Out_       ID3DX10DataProcessor **ppDataProcessor
);
```

Parameters

pFileName [in]

Type: **LPCSTR**

A string that contains the effect filename.

pDefines [in]

Type: **const D3D_SHADER_MACRO***

A NULL-terminated array of shader macros (see **D3D_SHADER_MACRO**); set this to **NULL** to specify no macros.

pInclude [in]

Type: **LPD3D10INCLUDE**

A pointer to an include interface (see **ID3D10Include Interface**); set this to **NULL** to specify there is no include file.

pProfile [in]

Type: **LPCSTR**

A string that specifies the shader profile or shader model.

Flags [in]

Type: **UINT**

HLSL compile options (see Shader Flags).

FXFlags [in]

Type: **UINT**

Effect compile options (see Compile and Effect Flags).

pDevice [in]

Type: **ID3D10Device***

A pointer to the device (see **ID3D10Device Interface**) that will use the resources.

ppErrorBuffer [out]

Type: **ID3D10Blob****

The address of a pointer to memory (see **ID3D10Blob Interface**) that contains effect compile errors, if there were any.

ppDataProcessor [out]

Type: **ID3DX10DataProcessor****

Address of a pointer to a buffer that contains the data processor created (see **ID3DX10DataProcessor Interface**).

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10Async.h

See also

General Purpose Functions

D3DX10CreateAsyncFileLoader function

Article • 03/15/2021 • 2 minutes to read

Create an asynchronous-file loader.

Syntax

```
C++

HRESULT D3DX10CreateAsyncFileLoader(
  _In_  LPCTSTR           pFileName,
  _Out_ ID3DX10DataLoader **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: **ID3DX10DataLoader****

Address of a pointer to the asynchronous-data loader (see **ID3DX10DataLoader Interface**).

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10Async.h

See also

General Purpose Functions

D3DX10CreateAsyncMemoryLoader function

Article • 03/15/2021 • 2 minutes to read

Create an asynchronous-memory loader.

Syntax

C++

```
HRESULT D3DX10CreateAsyncMemoryLoader(
  _In_  LPCVOID           pData,
  _In_  SIZE_T            cbData,
  _Out_ ID3DX10DataLoader **ppDataLoader
);
```

Parameters

pData [in]

Type: **LPCVOID**

Pointer to the data.

cbData [in]

Type: **SIZE_T**

Size of the data.

ppDataLoader [out]

Type: **ID3DX10DataLoader****

The address of a pointer to the asynchronous-data processor (see **ID3DX10DataProcessor Interface**).

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10Async.h

See also

General Purpose Functions

D3DX10CreateAsyncResourceLoader function

Article • 03/15/2021 • 2 minutes to read

Create an asynchronous-resource loader.

Syntax

C++

```
HRESULT D3DX10CreateAsyncResourceLoader(
  _In_  HMODULE           hSrcModule,
  _In_  LPCTSTR           pSrcResource,
  _Out_ ID3DX10DataLoader **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: **ID3DX10DataLoader****

The address of a pointer to the asynchronous-data processor (see **ID3DX10DataProcessor Interface**).

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10Async.h

See also

General Purpose Functions

D3DX10CreateAsyncShaderCompilerProcessor function

Article • 03/15/2021 • 2 minutes to read

Compile a shader and create a data processor asynchronously.

Syntax

C++

```
HRESULT D3DX10CreateAsyncShaderCompilerProcessor(
  _In_        LPCSTR                pFileName,
  _In_  const D3D_SHADER_MACRO   *pDefines,
  _In_        LPD3D10INCLUDE        pInclude,
  _In_        LPCSTR                pFunctionName,
  _In_        LPCSTR                pProfile,
  _In_        UINT                  Flags,
  _Out_       ID3D10Blob            **ppCompiledShader,
  _Out_       ID3D10Blob            **ppErrorBuffer,
  _Out_       ID3DX10DataProcessor **ppDataProcessor
);
```

Parameters

pFileName [in]

Type: **LPCSTR**

A string that contains the shader filename.

pDefines [in]

Type: **const D3D_SHADER_MACRO***

A NULL-terminated array of shader macros (see **D3D_SHADER_MACRO**); set this to **NULL** to specify no macros.

pInclude [in]

Type: **LPD3D10INCLUDE**

A pointer to an include interface (see **ID3D10Include Interface**); set this to **NULL** to specify there is no include file.

pFunctionName [in]

Type: **LPCSTR**

Name of the entry point function for the shader.

pProfile [in]

Type: **LPCSTR**

A string that specifies the shader profile or shader model.

Flags [in]

Type: **UINT**

HLSL compile options (see Shader Flags).

ppCompiledShader [out]

Type: **ID3D10Blob****

Address of a pointer to the compiled shader. See **ID3D10Blob Interface**.

ppErrorBuffer [out]

Type: **ID3D10Blob****

Address of a pointer to a buffer that contains compile errors (see **ID3D10Blob Interface**).

ppDataProcessor [out]

Type: **ID3DX10DataProcessor****

Address of a pointer to a buffer that contains the data processor created (see **ID3DX10DataProcessor Interface**).

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value

Requirement	Value
Header	D3DX10Async.h

See also

General Purpose Functions

D3DX10CreateAsyncShaderPreprocessProcessor function

Article • 03/15/2021 • 2 minutes to read

Create a data processor for a shader asynchronously.

Syntax

```
C++

HRESULT D3DX10CreateAsyncShaderPreprocessProcessor(
  _In_        LPCSTR                pFileName,
  _In_  const D3D_SHADER_MACRO   *pDefines,
  _In_        LPD3D10INCLUDE        pInclude,
  _Out_       ID3D10Blob            **ppShaderText,
  _Out_       ID3D10Blob            **ppErrorBuffer,
  _Out_       ID3DX10DataProcessor **ppDataProcessor
);
```

Parameters

pFileName [in]

Type: **LPCSTR**

A string that contains the shader filename.

pDefines [in]

Type: **const D3D_SHADER_MACRO***

A NULL-terminated array of shader macros (see **D3D_SHADER_MACRO**); set this to **NULL** to specify no macros.

pInclude [in]

Type: **LPD3D10INCLUDE**

A pointer to an include interface (see **ID3D10Include 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 (see **ID3D10Blob Interface**).

ppErrorBuffer [out]

Type: **ID3D10Blob****

Address of a pointer to a buffer that contains compile errors (see **ID3D10Blob Interface**).

ppDataProcessor [out]

Type: **ID3DX10DataProcessor****

Address of a pointer to a buffer that contains the data processor created (see **ID3DX10DataProcessor Interface**).

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10Async.h

See also

General Purpose Functions

D3DX10CreateAsyncShaderResourceViewProcessor function

Article • 03/15/2021 • 2 minutes to read

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 D3DX10 that uses **thread pumps**.

Syntax

```
C++

HRESULT D3DX10CreateAsyncShaderResourceViewProcessor(
  _In_  ID3D10Device           *pDevice,
  _In_  D3DX10_IMAGE_LOAD_INFO *pLoadInfo,
  _Out_ ID3DX10DataProcessor   **ppDataProcessor
);
```

Parameters

pDevice [in]

Type: **ID3D10Device***

Pointer to the Direct3D device (see **ID3D10Device Interface**) that will be used to create a resource and a shader-resource view for that resource.

pLoadInfo [in]

Type: **D3DX10_IMAGE_LOAD_INFO***

Optional. Identifies the characteristics of a texture (see **D3DX10_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: **ID3DX10DataProcessor****

Address of a pointer to a buffer that contains the data processor created (see **ID3DX10DataProcessor Interface**).

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10Async.h

See also

General Purpose Functions

D3DX10CreateDevice function

Article • 03/15/2021 • 2 minutes to read

Create the best Direct3D 10 device that represents the display adapter. If a Direct3D 10.1-compatible device can be created, it will be possible to acquire an **ID3D10Device1 Interface** pointer from the returned device interface pointer.

Syntax

C++

```
HRESULT D3DX10CreateDevice(
  _In_  IDXGIAdapter      *pAdapter,
  _In_  D3D10_DRIVER_TYPE DriverType,
  _In_  HMODULE           Software,
  _In_  UINT              Flags,
  _Out_ ID3D10Device      **ppDevice
);
```

Parameters

pAdapter [in]

Type: **IDXGIAdapter***

Pointer to the display adapter (see the **IDXGIAdapter** interface) when creating a hardware device; otherwise set this parameter to **NULL**. If **NULL** is specified when creating a hardware device, Direct3D will use the first adapter enumerated by the **IDXGIFactory** interface.

DriverType [in]

Type: **D3D10_DRIVER_TYPE**

The device-driver type (see the **D3D10_DRIVER_TYPE** enumeration). The driver type determines the type of device you will create.

Software [in]

Type: **HMODULE**

A handle to a loaded module that implements a software driver (such as D3D10Ref.dll). To get a handle, call the GetModuleHandle function.

Flags [in]

Type: **UINT**

Device creation flags (see the **D3D10_CREATE_DEVICE_FLAG** enumeration) that enable API layers. These flags can be bitwise OR'd together.

ppDevice [out]

Type: **ID3D10Device****

Address of a pointer to the device created (see the **ID3D10Device** interface).

Return value

Type: **HRESULT**

This function returns one of the following Direct3D 10 Return Codes.

Remarks

This function attempts to create the best device for the hardware. First, the function attempts to create a 10.1 device. If a 10.1 device cannot be created, the function attempts to create a 10.0 device. If neither device is successfully created, the function returns E_FAIL.

If your application needs to create only a 10.1 device, or a 10.0 device only, use the following functions instead:

- Use the **D3D10CreateDevice** function to create a Direct3D 10.0 device only.
- Use the **D3D10CreateDevice1** function to create a Direct3D 10.1 device only.
- Use the **D3DX10GetFeatureLevel1** function to get an **ID3D10Device1** interface pointer from an **ID3D10Device** interface pointer.

A Direct3D 10.1 device can only be created on computers running Windows Vista Service Pack 1 or later, and with Direct3D 10.1-compatible hardware installed. However, it is legal to call this function on computers running any version of Windows that has the D3DX10 DLL installed.

Requirements

Requirement	Value

Requirement	Value
Header	D3DX10Core.h

See also

General Purpose Functions

D3DX10CreateDeviceAndSwapChain function

Article • 03/15/2021 • 2 minutes to read

Create the best Direct3D device and a swap chain.

Syntax

C++

```
HRESULT D3DX10CreateDeviceAndSwapChain(
  _In_  IDXGIAdapter         *pAdapter,
  _In_  D3D10_DRIVER_TYPE    DriverType,
  _In_  HMODULE              Software,
  _In_  UINT                 Flags,
  _In_  DXGI_SWAP_CHAIN_DESC *pSwapChainDesc,
  _Out_ IDXGISwapChain       **ppSwapChain,
  _Out_ ID3D10Device         **ppDevice
);
```

Parameters

pAdapter [in]

Type: **IDXGIAdapter***

Pointer to a **IDXGIAdapter**.

DriverType [in]

Type: **D3D10_DRIVER_TYPE**

The type of driver for the device. See **D3D10_DRIVER_TYPE**.

Software [in]

Type: **HMODULE**

A handle to the DLL that implements a software rasterizer. Must be **NULL** if DriverType is non-software. The HMODULE of a DLL can be obtained with LoadLibrary, LoadLibraryEx, or GetModuleHandle.

Flags [in]

Type: **UINT**

Optional. Device creation flags (see **D3D10_CREATE_DEVICE_FLAG**) that enable API layers. These flags can be bitwise OR'd together.

pSwapChainDesc [in]

Type: **DXGI_SWAP_CHAIN_DESC***

Description of the swap chain. See **DXGI_SWAP_CHAIN_DESC**.

ppSwapChain [out]

Type: **IDXGISwapChain****

Address of a pointer to an **IDXGISwapChain**.

ppDevice [out]

Type: **ID3D10Device****

Address of a pointer to an **ID3D10Device Interface** that will receive the newly created device.

Return value

Type: **HRESULT**

This method returns one of the following Direct3D 10 Return Codes.

Remarks

To create the best device, this method implements more than one device creation option. First, the method attempts to create a 10.1 device (and swap chain). If that fails, the method attempts to create a 10.0 device. If that fails, the method will fail. If your application needs to create only a 10.1 device, or a 10.0 device only, use these APIs instead:

- Use **D3D10CreateDeviceAndSwapChain** to create a Direct3D 10.0 (only) device and swap chain.
- Use **D3D10CreateDeviceAndSwapChain1** to create a Direct3D 10.1 (only) device and swap chain.

This method requires Windows Vista Service Pack 1.

Requirements

Requirement	Value
Header	D3DX10Core.h

See also

General Purpose Functions

D3DX10CreateEffectFromFile function

Article • 03/15/2021 • 2 minutes to read

Create an effect from a file.

Syntax

```
C++

HRESULT D3DX10CreateEffectFromFile(
  _In_        LPCTSTR            pFileName,
  _In_  const D3D_SHADER_MACRO *pDefines,
  _In_        ID3D10Include      *pInclude,
  _In_        LPCSTR             pProfile,
  _In_        UINT               HLSLFlags,
  _In_        UINT               FXFlags,
  _In_        ID3D10Device       *pDevice,
  _In_        ID3D10EffectPool   *pEffectPool,
  _In_        ID3DX10ThreadPump  *pPump,
  _Out_       ID3D10Effect       **ppEffect,
  _Out_       ID3D10Blob         **ppErrors,
  _Out_       HRESULT            *pHResult
);
```

Parameters

pFileName [in]

Type: **LPCTSTR**

Name of the ASCII effect file. If the compiler settings require Unicode, the data type LPCTSTR resolves to LPCWSTR. Otherwise, the data type resolves to LPCSTR.

pDefines [in]

Type: **const D3D_SHADER_MACRO***

A NULL-terminated array of shader macros (see **D3D_SHADER_MACRO**); set this to **NULL** to specify no macros.

pInclude [in]

Type: **ID3D10Include***

A pointer to an include interface (see **ID3D10Include Interface**). This parameter can be **NULL**.

pProfile [in]

Type: **LPCSTR**

A string that specifies the shader profile, or shader model.

HLSLFlags [in]

Type: **UINT**

HLSL compile options (see D3D10_SHADER Constants).

FXFlags [in]

Type: **UINT**

Effect compile options (see Compile and Effect Flags).

pDevice [in]

Type: **ID3D10Device***

A pointer to the device (see **ID3D10Device Interface**) that will use the resources.

pEffectPool [in]

Type: **ID3D10EffectPool***

Pointer to an effect pool (see **ID3D10EffectPool Interface**) for sharing variables between effects.

pPump [in]

Type: **ID3DX10ThreadPump***

A pointer to a thread pump interface (see **ID3DX10ThreadPump Interface**). Use **NULL** to specify that this function should not return until it is completed.

ppEffect [out]

Type: **ID3D10Effect****

Address of a pointer to the effect (see **ID3D10Effect Interface**) that is created.

ppErrors [out]

Type: **ID3D10Blob****

The address of a pointer to memory (see **ID3D10Blob Interface**) that contains effect compile errors, if there were any.

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 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10Async.h

See also

General Purpose Functions

D3DX10CreateEffectFromMemory function

Article • 03/15/2021 • 2 minutes to read

Create an effect from memory.

Syntax

C++

```
HRESULT D3DX10CreateEffectFromMemory(
  _In_        LPCVOID            pData,
  _In_        SIZE_T             DataLength,
  _In_        LPCSTR             pSrcFileName,
  _In_  const D3D_SHADER_MACRO *pDefines,
  _In_        ID3D10Include      *pInclude,
  _In_        LPCSTR             pProfile,
  _In_        UINT               HLSLFlags,
  _In_        UINT               FXFlags,
  _In_        ID3D10Device       *pDevice,
  _In_        ID3D10EffectPool   *pEffectPool,
  _In_        ID3DX10ThreadPump  *pPump,
  _Out_       ID3D10Effect       **ppEffect,
  _Out_       ID3D10Blob         **ppErrors,
  _Out_       HRESULT            *pHResult
);
```

Parameters

pData [in]

Type: **LPCVOID**

Pointer to the effect in memory.

DataLength [in]

Type: **SIZE_T**

Size of the effect in memory.

pSrcFileName [in]

Type: **LPCSTR**

Name of the effect file in memory.

pDefines [in]

Type: **const** **D3D_SHADER_MACRO***

A NULL-terminated array of shader macros (see **D3D_SHADER_MACRO**); set this to **NULL** to specify no macros.

pInclude [in]

Type: **ID3D10Include***

A pointer to an include interface (see **ID3D10Include Interface**). This parameter can be **NULL**.

pProfile [in]

Type: **LPCSTR**

A string that specifies the shader profile, or shader model.

HLSLFlags [in]

Type: **UINT**

HLSL compile options (see D3D10_SHADER Constants).

FXFlags [in]

Type: **UINT**

Effect compile options (see D3D10_EFFECT Constants).

pDevice [in]

Type: **ID3D10Device***

A pointer to the device (see **ID3D10Device Interface**) that will use the resources.

pEffectPool [in]

Type: **ID3D10EffectPool***

Pointer to an effect pool (see **ID3D10EffectPool Interface**) for sharing variables between effects.

pPump [in]

Type: **ID3DX10ThreadPump***

A pointer to a thread pump interface (see **ID3DX10ThreadPump Interface**). Use **NULL** to specify that this function should not return until it is completed.

ppEffect [out]

Type: **ID3D10Effect****

Address of a pointer to the effect (see **ID3D10Effect Interface**) that is created.

ppErrors [out]

Type: **ID3D10Blob****

The address of a pointer to memory (see **ID3D10Blob Interface**) that contains effect compile errors, if there were any.

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 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10Async.h

See also

General Purpose Functions

D3DX10CreateEffectFromResource function

Article • 03/15/2021 • 2 minutes to read

Create an effect from a resource.

Syntax

C++

```
HRESULT D3DX10CreateEffectFromResource(
  _In_        HMODULE            hModule,
  _In_        LPCTSTR            pResourceName,
  _In_        LPCTSTR            pSrcFileName,
  _In_  const D3D10_SHADER_MACRO *pDefines,
  _In_        ID3D10Include      *pInclude,
  _In_        LPCSTR             pProfile,
  _In_        UINT               HLSLFlags,
  _In_        UINT               FXFlags,
  _In_        ID3D10Device       *pDevice,
  _In_        ID3D10EffectPool   *pEffectPool,
  _In_        ID3DX10ThreadPump  *pPump,
  _Out_       ID3D10Effect       **ppEffect,
  _Out_       ID3D10Blob         **ppErrors,
  _Out_       HRESULT            *pHResult
);
```

Parameters

hModule [in]

Type: **HMODULE**

A handle to the resource module containing the effect. HMODULE can be obtained with GetModuleHandle Function.

pResourceName [in]

Type: **LPCTSTR**

Name of the resource in hModule. 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 D3D_SHADER_MACRO***

A NULL-terminated array of shader macros (see **D3D_SHADER_MACRO**); set this to **NULL** to specify no macros.

pInclude [in]

Type: **ID3D10Include***

A pointer to an include interface (see **ID3D10Include Interface**). This parameter can be **NULL**.

pProfile [in]

Type: **LPCSTR**

A string that specifies the shader profile, or shader model.

HLSLFlags [in]

Type: **UINT**

HLSL compile options (see D3D10_SHADER Constants).

FXFlags [in]

Type: **UINT**

Effect compile options (see Compile and Effect Flags).

pDevice [in]

Type: **ID3D10Device***

A pointer to the device (see **ID3D10Device Interface**) that will use the resources.

pEffectPool [in]

Type: **ID3D10EffectPool***

Pointer to an effect pool (see **ID3D10EffectPool Interface**) for sharing variables between effects.

pPump [in]

Type: **ID3DX10ThreadPump***

A pointer to a thread pump interface (see **ID3DX10ThreadPump Interface**). Use **NULL** to specify that this function should not return until it is completed.

ppEffect [out]

Type: **ID3D10Effect****

Address of a pointer to the effect (see **ID3D10Effect Interface**) that is created.

ppErrors [out]

Type: **ID3D10Blob****

The address of a pointer to memory (see **ID3D10Blob Interface**) that contains effect compile errors, if there were any.

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 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10Async.h

See also

General Purpose Functions

D3DX10CreateEffectPoolFromFile function

Article • 03/15/2021 • 2 minutes to read

Create an effect pool from an effect file.

Syntax

C++

```
HRESULT D3DX10CreateEffectPoolFromFile(
  _In_        LPCTSTR            pFileName,
  _In_  const D3D_SHADER_MACRO *pDefines,
  _In_        ID3D10Include      *pInclude,
  _In_        LPCSTR             pProfile,
  _In_        UINT               HLSLFlags,
  _In_        UINT               FXFlags,
  _In_        ID3D10Device       *pDevice,
  _In_        ID3DX10ThreadPump  *pPump,
  _Out_       ID3D10EffectPool   **ppEffectPool,
  _Out_       ID3D10Blob         **ppErrors,
  _Out_       HRESULT            *pHResult
);
```

Parameters

pFileName [in]

Type: LPCTSTR

The effect filename. If the compiler settings require Unicode, the data type LPCTSTR resolves to LPCWSTR. Otherwise, the data type resolves to LPCSTR.

pDefines [in]

Type: **const** D3D_SHADER_MACRO*

A NULL-terminated array of shader macros (see D3D_SHADER_MACRO); set this to **NULL** to specify no macros.

pInclude [in]

Type: ID3D10Include*

A pointer to an include interface (see **ID3D10Include Interface**). This parameter can be **NULL**.

pProfile [in]

Type: **LPCSTR**

A string that specifies the shader profile, or shader model.

HLSLFlags [in]

Type: **UINT**

HLSL compile options (see D3D10_SHADER Constants).

FXFlags [in]

Type: **UINT**

Effect compile options (see Compile and Effect Flags).

pDevice [in]

Type: **ID3D10Device***

A pointer to the device (see **ID3D10Device Interface**) that will use the resources.

pPump [in]

Type: **ID3DX10ThreadPump***

A pointer to a thread pump interface (see **ID3DX10ThreadPump Interface**). Use **NULL** to specify that this function should not return until it is completed.

ppEffectPool [out]

Type: **ID3D10EffectPool****

The address of a pointer to the effect pool (see **ID3D10EffectPool Interface**).

ppErrors [out]

Type: **ID3D10Blob****

The address of a pointer to memory (see **ID3D10Blob Interface**) that contains effect compile errors, if there were any.

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 10 Return Codes.

Remarks

This example creates an effect pool from the effect used in the BasicHLSL10 Sample.

```
// Create an effect pool from an effect in memory
ID3D10EffectPool * l_pEffectPool = NULL;
ID3D10Blob* l_pBlob_Errors = NULL;
WCHAR str[MAX_PATH];
hr = DXUTFindDXSDKMediaFileCch( str, MAX_PATH, L"BasicHLSL10.fx" );
hr = D3DX10CreateEffectPoolFromFile( str,
    NULL, NULL, D3D10_SHADER_ENABLE_STRICTNESS,
    0, pd3dDevice, NULL, &l_pEffectPool,
    &l_pBlob_Errors );
```

Requirements

Requirement	Value
Header	D3DX10Async.h

See also

General Purpose Functions

D3DX10CreateEffectPoolFromMemory function

Article • 03/15/2021 • 2 minutes to read

Create an effect pool from an effect in memory.

Syntax

C++

```
HRESULT D3DX10CreateEffectPoolFromMemory(
  _In_        LPCVOID            pData,
  _In_        SIZE_T             DataLength,
  _In_        LPCSTR             pSrcFileName,
  _In_  const D3D_SHADER_MACRO *pDefines,
  _In_        ID3D10Include      *pInclude,
  _In_        LPCSTR             pProfile,
  _In_        UINT               HLSLFlags,
  _In_        UINT               FXFlags,
  _In_        ID3D10Device       *pDevice,
  _In_        ID3DX10ThreadPump  *pPump,
  _Out_       ID3D10EffectPool   **ppEffectPool,
  _Out_       ID3D10Blob         **ppErrors,
  _Out_       HRESULT            *pHResult
);
```

Parameters

pData [in]

Type: **LPCVOID**

A pointer to the effect.

DataLength [in]

Type: **SIZE_T**

The size of the effect.

pSrcFileName [in]

Type: **LPCSTR**

The name of the effect file.

pDefines [in]

Type: **const** **D3D_SHADER_MACRO***

A NULL-terminated array of shader macros (see **D3D_SHADER_MACRO**); set this to **NULL** to specify no macros.

pInclude [in]

Type: **ID3D10Include***

A pointer to an include interface (see **ID3D10Include Interface**). This parameter can be **NULL**.

pProfile [in]

Type: **LPCSTR**

A string that specifies the shader profile, or shader model.

HLSLFlags [in]

Type: **UINT**

HLSL compile options (see D3D10_SHADER Constants).

FXFlags [in]

Type: **UINT**

Effect compile options (see Compile and Effect Flags).

pDevice [in]

Type: **ID3D10Device***

A pointer to the device (see **ID3D10Device Interface**) that will use the resources.

pPump [in]

Type: **ID3DX10ThreadPump***

A pointer to a thread pump interface (see **ID3DX10ThreadPump Interface**). Use **NULL** to specify that this function should not return until it is completed.

ppEffectPool [out]

Type: **ID3D10EffectPool****

The address of a pointer to the effect pool (see **ID3D10EffectPool Interface**).

ppErrors [out]

Type: **ID3D10Blob****

The address of a pointer to memory (see **ID3D10Blob Interface**) that contains effect compile errors, if there were any.

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**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10Async.h

See also

General Purpose Functions

D3DX10CreateEffectPoolFromResource function

Article • 03/15/2021 • 2 minutes to read

Create an effect pool from a resource.

Syntax

C++

```
HRESULT D3DX10CreateEffectPoolFromResource(
  _In_        HMODULE            hModule,
  _In_        LPCTSTR            pResourceName,
  _In_        LPCTSTR            pSrcFileName,
  _In_  const D3D_SHADER_MACRO *pDefines,
  _In_        ID3D10Include      *pInclude,
  _In_        LPCSTR             pProfile,
  _In_        UINT               HLSLFlags,
  _In_        UINT               FXFlags,
  _In_        ID3D10Device       *pDevice,
  _In_        ID3DX10ThreadPump  *pPump,
  _In_        ID3D10EffectPool   **ppEffectPool,
  _In_        ID3D10Blob         **ppErrors,
  _Out_       HRESULT            *pHResult
);
```

Parameters

hModule [in]

Type: **HMODULE**

A handle to the resource module containing the effect. HMODULE can be obtained with GetModuleHandle Function.

pResourceName [in]

Type: **LPCTSTR**

The name of the resource in hModule. 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 D3D_SHADER_MACRO***

A NULL-terminated array of shader macros (see **D3D_SHADER_MACRO**); set this to **NULL** to specify no macros.

pInclude [in]

Type: **ID3D10Include***

A pointer to an include interface (see **ID3D10Include Interface**). This parameter can be **NULL**.

pProfile [in]

Type: **LPCSTR**

A string that specifies the shader profile, or shader model.

HLSLFlags [in]

Type: **UINT**

HLSL compile options (see D3D10_SHADER Constants).

FXFlags [in]

Type: **UINT**

Effect compile options (see Compile and Effect Flags).

pDevice [in]

Type: **ID3D10Device***

A pointer to the device (see **ID3D10Device Interface**) that will use the resources.

pPump [in]

Type: **ID3DX10ThreadPump***

A pointer to a thread pump interface (see **ID3DX10ThreadPump Interface**). Use **NULL** to specify that this function should not return until it is completed.

ppEffectPool [in]

Type: **ID3D10EffectPool****

The address of a pointer to the effect pool (see **ID3D10EffectPool Interface**).

ppErrors [in]

Type: **ID3D10Blob****

The address of a pointer to memory (see **ID3D10Blob Interface**) that contains effect compile errors, if there were any.

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 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10Async.h

See also

General Purpose Functions

D3DX10CreateFont function

Article • 03/15/2021 • 2 minutes to read

Creates a font object for a device and font.

> **Note**
>
> Instead of using this function, we recommend that you use **DirectWrite** and the **DirectXTK** library, **SpriteFont** class.

Syntax

C++

```
HRESULT D3DX10CreateFont(
  _In_  ID3D10Device *pDevice,
  _In_  INT          Height,
  _In_  UINT         Width,
  _In_  UINT         Weight,
  _In_  UINT         MipLevels,
  _In_  BOOL         Italic,
  _In_  UINT         CharSet,
  _In_  UINT         OutputPrecision,
  _In_  UINT         Quality,
  _In_  UINT         PitchAndFamily,
  _In_  LPCTSTR      pFaceName,
  _Out_ LPD3DX10FONT *ppFont
);
```

Parameters

pDevice [in]

Type: **ID3D10Device***

Pointer to an ID3D10Device interface, the device to be associated with the font object.

Height [in]

Type: **INT**

The height of the characters in logical units.

Width [in]

Type: **UINT**

The width of the characters in logical units.

Weight [in]

Type: **UINT**

Typeface weight. One example is bold.

MipLevels [in]

Type: **UINT**

The number of mipmap levels.

Italic [in]

Type: **BOOL**

True for italic font, false otherwise.

CharSet [in]

Type: **UINT**

The character set of the font.

OutputPrecision [in]

Type: **UINT**

Specifies how Windows should attempt to match the desired font sizes and characteristics with actual fonts. Use OUT_TT_ONLY_PRECIS for instance, to ensure that you always get a TrueType font.

Quality [in]

Type: **UINT**

Specifies how Windows should match the desired font with a real font. It applies to raster fonts only and should not affect TrueType fonts.

PitchAndFamily [in]

Type: **UINT**

Pitch and family index.

pFaceName [in]

Type: **LPCTSTR**

String containing the typeface name. If the compiler settings require Unicode, the data type LPCTSTR resolves to LPCWSTR. Otherwise, the data type resolves to LPCSTR. See Remarks.

ppFont [out]

Type: **LPD3DX10FONT***

Returns a pointer to an ID3DX10Font interface, representing the created font object.

Return value

Type: **HRESULT**

If the function succeeds, the return value is S_OK. If the function fails, the return value can be one of the following: D3DERR_INVALIDCALL, D3DXERR_INVALIDDATA, E_OUTOFMEMORY.

Remarks

The compiler setting also determines the function version. If Unicode is defined, the function call resolves to D3DXCreateFontW. Otherwise, the function call resolves to D3DXCreateFontA because ANSI strings are being used.

If you want more information about font parameters, see The Logical Font.

Requirements

Requirement	Value
Header	D3DX10Core.h
Library	D3DX10.lib

See also

General Purpose Functions

D3DX10CreateFontIndirect function

Article • 03/15/2021 • 2 minutes to read

Creates a font object.

> **Note**
>
> Instead of using this function, we recommend that you use **DirectWrite** and the **DirectXTK** library, **SpriteFont** class.

Syntax

C++

```
HRESULT D3DX10CreateFontIndirect(
  _In_        ID3D10Device     *pDevice,
  _In_  const D3DX10_FONT_DESC *pDesc,
  _Out_       LPD3DX10FONT     *ppFont
);
```

Parameters

pDevice [in]

Type: **ID3D10Device***

Pointer to an **ID3D10Device Interface** interface.

pDesc [in]

Type: **const D3DX10_FONT_DESC***

Pointer to a **D3DX10_FONT_DESC** structure, describing the attributes of the font object to create. If Unicode is defined, the function call resolves to D3DXCreateFontIndirectW. Otherwise, the function call resolves to D3DXCreateFontIndirectA because ANSI strings are being used.

ppFont [out]

Type: **LPD3DX10FONT***

Returns a pointer to an **ID3DX10Font Interface**.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10Core.h
Library	D3DX10.lib

See also

General Purpose Functions

D3DX10CreateSprite function

Article • 03/15/2021 • 2 minutes to read

Create a sprite for drawing a 2D texture.

> **Note**
>
> Instead of using this function, we recommend that you use **Direct2D** and the **DirectXTK** library, **SpriteBatch** class.

Syntax

C++

```
HRESULT D3DX10CreateSprite(
  _In_  ID3D10Device   *pDevice,
  _In_  UINT           cDeviceBufferSize,
  _Out_ LPD3DX10SPRITE *ppSprite
);
```

Parameters

pDevice [in]

Type: **ID3D10Device***

A pointer to the device (see **ID3D10Device Interface**) that will draw the sprite.

cDeviceBufferSize [in]

Type: **UINT**

The size of the vertex buffer, in number of sprites, that will be sent to the device when **ID3DX10Sprite::Flush** or **ID3DX10Sprite::DrawSpritesImmediate** is called. This should be a small number if you know you will be rendering a small number of sprites at a time (to save memory) and a large number if you know you will be rendering a large number of sprites at a time. The maximum value is 4096. If 0 is specified, the vertex buffer size will automatically be set to 4096.

ppSprite [out]

Type: **LPD3DX10SPRITE***

The address of a pointer to a sprite interface (see **ID3DX10Sprite Interface**).

Return value

Type: **HRESULT**

If the function succeeds, the return value is S_OK. If the function fails, the return value can be one of the following: D3DERR_INVALIDCALL, E_OUTOFMEMORY.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

General Purpose Functions

D3DX10CreateThreadPump function

Article • 03/15/2021 • 2 minutes to read

Create a thread pump.

Syntax

```
C++

HRESULT D3DX10CreateThreadPump(
  _In_  UINT              cIoThreads,
  _In_  UINT              cProcThreads,
  _Out_ ID3DX10ThreadPump **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: **ID3DX10ThreadPump****

The created thread pump. See **ID3DX10ThreadPump Interface**.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Remarks

A thread pump is a very resource-intensive object. Only one thread pump should be created per application.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

General Purpose Functions

D3DX10DebugMute function

Article • 03/15/2021 • 2 minutes to read

Enable or disable debug messages.

Syntax

```
C++

HRESULT D3DX10DebugMute(
  _In_ BOOL Mute
);
```

Parameters

Mute [in]

Type: **BOOL**

Set to **TRUE** to enable debug messages; set to **FALSE** to disable debug messages.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Remarks

Use this function to disable error messages for D3DX10 APIs during debug; the use of this function should be guarded by the D3D10_DEBUG compiler switch.

```
#ifdef D3D10_DEBUG
  BOOL WINAPI D3DX10DebugMute(BOOL Mute);
#endif
```

The default state is **TRUE** for a debug build.

Requirements

Requirement	Value
Header	D3DX10Core.h
Library	D3DX10.lib

See also

General Purpose Functions

D3DX10DisassembleEffect function

Article • 03/15/2021 • 2 minutes to read

> **Note**
>
> Instead of using this legacy function, we recommend that you use the **D3DDisassemble** API.

This function -- which disassembles a compiled effect into a text string that contains assembly instructions and register assignments -- has been deprecated. Instead, use **D3DDisassemble10Effect**.

Syntax

```
C++

HRESULT D3DX10DisassembleEffect(
  _In_  ID3D10Effect *pEffect,
  _In_  BOOL         EnableColorCode,
  _Out_ ID3D10Blob   **ppDisassembly
);
```

Parameters

pEffect [in]

Type: **ID3D10Effect***

A pointer to the effect interface (see **ID3D10Effect Interface**).

EnableColorCode [in]

Type: **BOOL**

Include HTML tags in the output to color code the result.

ppDisassembly [out]

Type: **ID3D10Blob****

Address of a buffer (see **ID3D10Blob Interface**) which contains the disassembled effect.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10Core.h

See also

General Purpose Functions

D3DX10DisassembleShader function

Article • 03/15/2021 • 2 minutes to read

> **Note**
>
> Instead of using this legacy function, we recommend that you use the **D3DDisassemble** API.

This function -- which disassembles a compiled shader into a text string that contains assembly instructions and register assignments -- no longer exists. Instead, use **D3DDisassemble10Effect**.

Syntax

```
C++

HRESULT D3DX10DisassembleShader(
  _In_  const void       *pShader,
  _In_        SIZE_T     BytecodeLength,
  _In_        BOOL       EnableColorCode,
  _In_        LPCSTR     pComments,
  _Out_       ID3D10Blob **ppDisassembly
);
```

Parameters

pShader [in]

Type: **const void***

A pointer to the **compiled shader**.

BytecodeLength [in]

Type: **SIZE_T**

The size of pShader.

EnableColorCode [in]

Type: **BOOL**

Include HTML tags in the output to color code the result.

pComments [in]

Type: **LPCSTR**

The comment string at the top of the shader that identifies the shader constants and variables.

ppDisassembly [out]

Type: **ID3D10Blob****

Address of a buffer (see **ID3D10Blob Interface**) which contains the disassembled shader.

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Remarks

This returned text includes a header with the version of the HLSL compiler used to generate this object, comments describing the memory layout of the constant buffers used by the shader, input and output signatures, and resource binding points.

Here is an example of disassembling a compiled shader. The example assumes you start with a compiled shader (shown as *pVSBuf* which you can see in HLSLWithoutFX10 Sample).

```
LPCSTR commentString = NULL;
ID3D10Blob* pIDisassembly = NULL;
char* pDisassembly = NULL;
if( pVSBuf )
{
    D3D10DisassembleShader( (UINT*) pVSBuf->GetBufferPointer(),
        pVSBuf->GetBufferSize(), TRUE, commentString, &pIDisassembly );
    if( pIDisassembly )
    {
        FILE* pFile = fopen( "shader.htm", "w" );
        if( pFile)
        {
            fputs( (char*)pIDisassembly->GetBufferPointer(), pFile );
            fclose( pFile );
```

```
        }
    }
}
```

Requirements

Requirement	Value
Header	D3DX10Core.h

See also

General Purpose Functions

D3DX10GetDriverLevel

Article • 01/06/2021 • 2 minutes to read

This function has been deprecated, use **D3DX10GetFeatureLevel1** instead.

Related topics

General Purpose Functions

D3DX10GetFeatureLevel1 function

Article • 03/15/2021 • 2 minutes to read

Get a Direct3D 10.1 device interface pointer from a Direct3D 10.0 interface pointer.

Syntax

```C++
HRESULT D3DX10GetFeatureLevel1(
  _In_  ID3D10Device  *pDevice,
  _Out_ ID3D10Device1 **ppDevice
);
```

Parameters

pDevice [in]

Type: **ID3D10Device***

Pointer to the Direct3D 10.0 device (see the **ID3D10Device** interface).

ppDevice [out]

Type: **ID3D10Device1****

Pointer to the Direct3D 10.1 device (see the **ID3D10Device1** interface).

Return value

Type: **HRESULT**

This function returns one of the following Direct3D 10 Return Codes. If a Direct3D 10.1 device interface can be acquired, this function succeeds and passes a pointer to the 10.1 interface using the *ppDevice* parameter. If a Direct3D 10.1 device interface cannot be acquired, this function returns E_FAIL, and will not return anything for the *ppDevice* parameter.

Remarks

For this function to succeed, you must have acquired the supplied **ID3D10Device** pointer using a call to the **D3DX10CreateDevice** function, the **D3DX10CreateDeviceAndSwapChain** function, the **D3D10CreateDevice1** function, or the **D3D10CreateDeviceAndSwapChain1** function.

You can only create a Direct3D 10.1 device on computers running Windows Vista Service Pack 1 or later, and with Direct3D 10.1-compatible hardware installed. This function will return E_FAIL on any computer not meeting these requirements. However, you can call this function on any version of Windows that has the D3DX10 DLL installed.

Requirements

Requirement	Value
Header	D3DX10Core.h

See also

General Purpose Functions

D3DX10PreprocessShaderFromFile function

Article • 03/15/2021 • 2 minutes to read

> **Note**
>
> Instead of using this legacy function, we recommend that you use the **D3DPreprocess** API.

Create a shader from a file without compiling it.

Syntax

C++

```
HRESULT D3DX10PreprocessShaderFromFile(
  _In_        LPCTSTR            pFileName,
  _In_  const D3D10_SHADER_MACRO *pDefines,
  _In_        LPD3D10INCLUDE     pInclude,
  _In_        ID3DX10ThreadPump  *pPump,
  _Out_       ID3D10Blob         **ppShaderText,
  _Out_       ID3D10Blob         **ppErrorMsgs
);
```

Parameters

pFileName [in]

Type: **LPCTSTR**

Name of the shader file.

pDefines [in]

Type: **const D3D_SHADER_MACRO***

A NULL-terminated array of shader macros (see **D3D_SHADER_MACRO**); set this to **NULL** to specify no macros.

pInclude [in]

Type: **LPD3D10INCLUDE**

A pointer to an include interface (see **ID3D10Include Interface**); set this to **NULL** to specify there is no include file.

pPump [in]

Type: **ID3DX10ThreadPump***

A pointer to a thread pump interface (see **ID3DX10ThreadPump Interface**). Use **NULL** to specify that this function should not return until it is completed.

ppShaderText [out]

Type: **ID3D10Blob****

A pointer to memory (see **ID3D10Blob Interface**) that contains the uncompiled shader.

ppErrorMsgs [out]

Type: **ID3D10Blob****

The address of a pointer to memory (see **ID3D10Blob Interface**) that contains effect-creation errors, if any occurred.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10Async.h

See also

General Purpose Functions

D3DX10PreprocessShaderFromMemory function

Article • 03/15/2021 • 2 minutes to read

> **Note**
>
> Instead of using this legacy function, we recommend that you use the **D3DPreprocess** API.

Create a shader from memory without compiling it.

Syntax

C++

```
HRESULT D3DX10PreprocessShaderFromMemory(
  _In_        LPCSTR             pSrcData,
  _In_        SIZE_T             SrcDataSize,
  _In_        LPCSTR             pFileName,
  _In_  const D3D_SHADER_MACRO *pDefines,
  _In_        LPD3D10INCLUDE     pInclude,
  _In_        ID3DX10ThreadPump  *pPump,
  _Out_       ID3D10Blob         **ppShaderText,
  _Out_       ID3D10Blob         **ppErrorMsgs
);
```

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 D3D_SHADER_MACRO***

A NULL-terminated array of shader macros (see **D3D_SHADER_MACRO**); set this to **NULL** to specify no macros.

pInclude [in]

Type: **LPD3D10INCLUDE**

A pointer to an include interface (see **ID3D10Include Interface**); set this to **NULL** to specify there is no include file.

pPump [in]

Type: **ID3DX10ThreadPump***

A pointer to a thread pump interface (see **ID3DX10ThreadPump Interface**). Use **NULL** to specify that this function should not return until it is completed.

ppShaderText [out]

Type: **ID3D10Blob****

A pointer to memory (see **ID3D10Blob Interface**) that contains the uncompiled shader.

ppErrorMsgs [out]

Type: **ID3D10Blob****

The address of a pointer to memory (see **ID3D10Blob Interface**) that contains effect-creation errors, if any occurred.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

General Purpose Functions

D3DX10PreprocessShaderFromResource function

Article • 03/15/2021 • 2 minutes to read

> **Note**
>
> Instead of using this legacy function, we recommend that you use the **D3DPreprocess** API.

Create a shader from a resource without compiling it.

Syntax

C++

```
HRESULT D3DX10PreprocessShaderFromResource(
  _In_        HMODULE            hModule,
  _In_        LPCTSTR            pResourceName,
  _In_        LPCTSTR            pSrcFileName,
  _In_  const D3D_SHADER_MACRO *pDefines,
  _In_        LPD3D10INCLUDE     pInclude,
  _In_        ID3DX10ThreadPump  *pPump,
  _Out_       ID3D10Blob         **ppShaderText,
  _Out_       ID3D10Blob         **ppErrorMsgs
);
```

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 D3D_SHADER_MACRO***

A NULL-terminated array of shader macros (see **D3D_SHADER_MACRO**); set this to **NULL** to specify no macros.

pInclude [in]

Type: **LPD3D10INCLUDE**

A pointer to an include interface (see **ID3D10Include Interface**); set this to **NULL** to specify there is no include file.

pPump [in]

Type: **ID3DX10ThreadPump***

A pointer to a thread pump interface (see **ID3DX10ThreadPump Interface**). Use **NULL** to specify that this function should not return until it is completed.

ppShaderText [out]

Type: **ID3D10Blob****

A pointer to memory (see **ID3D10Blob Interface**) that contains the uncompiled shader.

ppErrorMsgs [out]

Type: **ID3D10Blob****

The address of a pointer to memory (see **ID3D10Blob Interface**) that contains effect-creation errors, if any occurred.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

General Purpose Functions

D3DX10ReflectShader function

Article • 03/15/2021 • 2 minutes to read

This function -- which creates a shader-reflection object for retrieving information about a compiled shader -- no longer exists. Instead, use **D3DReflect** or **D3D11Reflect**.

Syntax

C++

```
HRESULT D3DX10ReflectShader(
  _In_  const void                    *pShaderBytecode,
  _In_        SIZE_T                  BytecodeLength,
  _Out_       ID3D10ShaderReflection1 **ppReflector
);
```

Parameters

pShaderBytecode [in]

Type: **const void***

A pointer to the compiled shader. To get this pointer see Getting a Pointer to a Compiled Shader.

BytecodeLength [in]

Type: **SIZE_T**

Length of pShaderBytecode.

ppReflector [out]

Type: **ID3D10ShaderReflection1****

Address of a shader reflection interface (see **ID3D10ShaderReflection1 Interface**.)

Return value

Type: **HRESULT**

Returns one of the following Direct3D 10 Return Codes.

Remarks

Here is an example of creating a shader-reflection object. The example assumes you start with a compiled shader (shown as

```
pVSBuf
```

which you can see in HLSLWithoutFX10 Sample).

```
ID3D10ShaderReflection1* pIShaderReflection1 = NULL;
D3D10_SHADER_DESC desc;
hr = D3D10ReflectShader( (void*) pVSBuf->GetBufferPointer(), pVSBuf-
>GetBufferSize(),
    &pIShaderReflection1 );
if( pIShaderReflection1 )
{
    pIShaderReflection1->GetDesc( &desc );
}
```

Requirements

Requirement	Value
Header	D3DX10Core.h

See also

General Purpose Functions

D3DX10UnsetAllDeviceObjects function

Article • 03/15/2021 • 2 minutes to read

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

C++

```
HRESULT D3DX10UnsetAllDeviceObjects(
  _In_ ID3D10Device *pDevice
);
```

Parameters

pDevice [in]

Type: **ID3D10Device***

Pointer to the device. See **ID3D10Device Interface**.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10Core.h
Library	D3DX10.lib

See also

General Purpose Functions

Math Functions (Direct3D 10 Graphics)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

The math library provided by the D3DX utility library supplies functions to compute 3D mathematical operations. Each of the functions can take the same object as the passed [in] and returned [out] parameters. Also, out parameters are typically returned as return values, so that the output of one math function can be used as a parameter for another math function.

- D3DXColorAdjustContrast
- D3DXColorAdjustSaturation
- D3DXCreateMatrixStack
- D3DXCpuOptimizations
- D3DXFloat16To32Array
- D3DXFloat32To16Array
- D3DXFresnelTerm
- D3DXMatrixAffineTransformation
- D3DXMatrixAffineTransformation2D
- D3DXMatrixDecompose
- D3DXMatrixDeterminant
- D3DXMatrixInverse
- D3DXMatrixLookAtLH
- D3DXMatrixLookAtRH
- D3DXMatrixMultiply
- D3DXMatrixMultiplyTranspose
- D3DXMatrixOrthoLH
- D3DXMatrixOrthoRH
- D3DXMatrixOrthoOffCenterLH
- D3DXMatrixOrthoOffCenterRH
- D3DXMatrixPerspectiveFovLH
- D3DXMatrixPerspectiveFovRH
- D3DXMatrixReflect
- D3DXMatrixRotationAxis

- D3DXMatrixRotationX
- D3DXMatrixRotationY
- D3DXMatrixRotationZ
- D3DXMatrixRotationYawPitchRoll
- D3DXMatrixScaling
- D3DXMatrixShadow
- D3DXMatrixTransformation
- D3DXMatrixTransformation2D
- D3DXMatrixTranslation
- D3DXMatrixTranspose
- D3DXPlaneFromPointNormal
- D3DXPlaneFromPoints
- D3DXPlaneIntersectLine
- D3DXPlaneNormalize
- D3DXPlaneTransform
- D3DXPlaneTransformArray
- D3DXQuaternionBaryCentric
- D3DXQuaternionExp
- D3DXQuaternionInverse
- D3DXQuaternionLn
- D3DXQuaternionMultiply
- D3DXQuaternionNormalize
- D3DXQuaternionRotationAxis
- D3DXQuaternionRotationMatrix
- D3DXQuaternionRotationYawPitchRoll
- D3DXQuaternionSlerp
- D3DXQuaternionSquad
- D3DXQuaternionSquadSetup
- D3DXQuaternionToAxisAngle
- D3DXSHEvalConeLight
- D3DXSHEvalDirection
- D3DXSHEvalDirectionalLight
- D3DXSHEvalHemisphereLight
- D3DXSHMultiply2
- D3DXSHMultiply3
- D3DXSHMultiply4
- D3DXSHMultiply5
- D3DXSHMultiply6
- D3DX10SHProjectCubeMap
- D3DXSHRotate

- D3DXSHRotateZ
- D3DXSHScale
- D3DXVec2BaryCentric
- D3DXVec2CatmullRom
- D3DXVec2Hermite
- D3DXVec2Normalize
- D3DXVec2Transform
- D3DXVec2TransformArray
- D3DXVec2TransformCoord
- D3DXVec2TransformCoordArray
- D3DXVec2TransformNormal
- D3DXVec2TransformNormalArray
- D3DXVec3BaryCentric
- D3DXVec3CatmullRom
- D3DXVec3Hermite
- D3DXVec3Normalize
- D3DXVec3Project
- D3DXVec3ProjectArray
- D3DXVec3Transform
- D3DXVec3TransformArray
- D3DXVec3TransformCoord
- D3DXVec3TransformCoordArray
- D3DXVec3TransformNormal
- D3DXVec3TransformNormalArray
- D3DXVec4BaryCentric
- D3DXVec4CatmullRom
- D3DXVec4Cross
- D3DXVec4Hermite
- D3DXVec4Normalize
- D3DXVec4Transform
- D3DXVec4TransformArray

Resolving Link Errors with D3DX Math Functions

The D3DX math functions are implemented identically in D3DX10 (D3DX10math.h) and D3DX9 (D3DX9math.h). This can cause link errors if a project implements both DirectX 9 and DirectX 10 code, and attempts to link a function from one header with the opposite library.

To eliminate the problem of including both headers, D3DX10math.h includes the following #define:

```
#ifndef __D3DX9MATH_H__
#define __D3DX9MATH_H__
```

To eliminate possible link errors, the DX SDK samples link to D3DX9 libraries first (D3DX9d.lib and D3DX9.lib) and then the D3DX10 libraries second (D3DX10d.lib and D3DX10.lib). These settings are under Project/Properties if you are using Visual Studio.

Related topics

D3DX Functions

D3DXColorAdjustContrast function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Adjusts the contrast value of a color.

Syntax

```C++
D3DXCOLOR* D3DXColorAdjustContrast(
  _In_       D3DXCOLOR *pOut,
  _In_ const D3DXCOLOR *pC,
  _In_       FLOAT     c
);
```

Parameters

pOut [in]

Type: **D3DXCOLOR***

[in, out] Pointer to a **D3DXCOLOR** that is the result of the operation.

pC [in]

Type: **const D3DXCOLOR***

Pointer to a source D3DXCOLOR structure.

c [in]

Type: **FLOAT**

Contrast value. This parameter linearly interpolates between fifty percent gray and the color, pC. There are no limits on the value of c. If this parameter is zero, then the

returned color is fifty percent gray. If this parameter is 1, then the returned color is the original color.

Return value

Type: D3DXCOLOR*

This function returns a pointer to a D3DXCOLOR structure that is the result of the contrast adjustment.

Remarks

The input alpha channel is copied, unmodified, to the output alpha channel.

The return value for this function is the same value returned in the pOut parameter. In this way, this function can be used as a parameter for another function.

This function interpolates the red, green, and blue color components of a D3DXCOLOR structure between fifty percent gray and a specified contrast value, as shown in the following example.

```
pOut->r = 0.5f + c * (pC->r - 0.5f);
```

If c is greater than 0 and less than 1, the contrast is decreased. If c is greater than 1, the contrast is increased.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXColorAdjustSaturation function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Adjusts the saturation value of a color.

Syntax

```
C++

D3DXCOLOR* D3DXColorAdjustSaturation(
  _In_       D3DXCOLOR *pOut,
  _In_ const D3DXCOLOR *pC,
  _In_       FLOAT     s
);
```

Parameters

pOut [in]

Type: **D3DXCOLOR***

Pointer to a **D3DXCOLOR** that is the result of the operation.

pC [in]

Type: **const D3DXCOLOR***

Pointer to a source D3DXCOLOR structure.

s [in]

Type: **FLOAT**

Saturation value. This parameter linearly interpolates between the color converted to grayscale and the original color, pC. There are no limits on the value of s. If s is 0, the

returned color is the grayscale color. If s is 1, the returned color is the original color.

Return value

Type: **D3DXCOLOR***

This function returns a pointer to a D3DXCOLOR structure that is the result of the saturation adjustment.

Remarks

The input alpha channel is copied, unmodified, to the output alpha channel.

The return value for this function is the same value returned in the pOut parameter. In this way, this function can be used as a parameter for another function.

This function interpolates the red, green, and blue color components of a D3DXCOLOR structure between an unsaturated color and a color, as shown in the following example.

```
//Approximate values for each component's contribution to luminance.
//Based upon the NTSC standard described in ITU-R Recommendation BT.709.
FLOAT grey = pC->r * 0.2125f + pC->g * 0.7154f + pC->b * 0.0721f;

pOut->r = grey + s * (pC->r - grey);
```

If s is greater than 0 and less than 1, the saturation is decreased. If s is greater than 1, the saturation is increased.

The grayscale color is computed as:

```
r = g = b = 0.2125*r + 0.7154*g + 0.0721*b;
```

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXCreateMatrixStack function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead along with this header from **GitHub**.

Create a matrix stack.

Syntax

C++

```
HRESULT D3DXCreateMatrixStack(
  _In_  UINT              Flags,
  _Out_ LPD3DXMATRIXSTACK *ppStack
);
```

Parameters

Flags [in]

Type: **UINT**

Not implemented. Specify zero.

ppStack [out]

Type: **LPD3DXMATRIXSTACK***

The address of a pointer to a matrix stack (see **ID3DXMatrixStack Interface**).

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXCpuOptimizations function

Article • 03/15/2021 • 2 minutes to read

Enables or disables CPU optimizations.

Syntax

```
C++

D3DX_CPU_OPTIMIZATION D3DXCpuOptimizations(
  _In_ BOOL Enable
);
```

Parameters

Enable [in]

Type: **BOOL**

TRUE to enable CPU optimizations; otherwise **FALSE**.

Return value

Type: **D3DX_CPU_OPTIMIZATION**

Returns the type of CPU detected, and for which optimizations exist (see **D3DX_CPU_OPTIMIZATION**).

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXFloat16To32Array function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Converts an array of 16-bit floats to 32-bit floats.

Syntax

```
C++

FLOAT* D3DXFloat16To32Array(
  _In_       FLOAT       *pOut,
  _In_ const D3DXFLOAT16 *pIn,
  _In_       UINT        n
);
```

Parameters

pOut [in]

Type: **FLOAT***

Pointer to the array of 32-bit floats.

pIn [in]

Type: **const D3DXFLOAT16***

Pointer to an array of 16-bit floats.

n [in]

Type: **UINT**

Number of elements in the array.

Return value

Type: **FLOAT***

Pointer to an array of 32-bit floats.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXFloat32To16Array function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Converts an array of 32-bit floats to 16-bit floats.

Syntax

```C++
D3DXFLOAT16* D3DXFloat32To16Array(
  _In_       D3DXFLOAT16 *pOut,
  _In_ const FLOAT       *pIn,
  _In_       UINT        n
);
```

Parameters

pOut [in]

Type: **D3DXFLOAT16***

Pointer to the array of 16-bit floats.

pIn [in]

Type: **const FLOAT***

Pointer to an array of 32-bit floats.

n [in]

Type: **UINT**

The number of elements in the array.

Return value

Type: **D3DXFLOAT16***

Pointer to an array of 16-bit floats.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXFresnelTerm function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX utility library is deprecated. We recommend that you use **DirectXMath** instead.

Calculate the Fresnel term.

Syntax

```C++
FLOAT D3DXFresnelTerm(
  _In_ FLOAT CosTheta,
  _In_ FLOAT RefractionIndex
);
```

Parameters

CosTheta [in]

Type: **FLOAT**

The value must be between 0 and 1.

RefractionIndex [in]

Type: **FLOAT**

The refraction index of a material. The value must be greater than 1.

Return value

Type: **FLOAT**

This function returns the Fresnel term for unpolarized light. CosTheta is the cosine of the incident angle.

Remarks

To find the Fresnel term (F):

If A is angle of incidence and B is the angle of refraction, then

```
F = 0.5 * [tan2(A - B) / tan2(A + B) + sin2(A - B) / sin2(A + B)]
  = 0.5 * sin2(A - B) / sin2(A + B) * [cos2(A + B) / cos2(A - B) + 1]

Let r   = sina(A) / sin(B)      (the relative refractive index)
Let c   = cos(A)
Let g   = (r2 + c2 - 1)1/2
```

Then, expanding using the trig identities and simplifying, you get:

```
F = 0.5 * (g + c)2 / (g - c)2 * ([c(g + c) - 1]2 / [c(g - c) + 1]2 + 1)
```

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXMatrixAffineTransformation function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX utility library is deprecated. We recommend that you use **DirectXMath** instead.

Builds a 3D affine transformation matrix. **NULL** arguments are treated as identity transformations.

Syntax

```C++
D3DXMATRIX* D3DXMatrixAffineTransformation(
  _In_       D3DXMATRIX     *pOut,
  _In_       FLOAT          Scaling,
  _In_ const D3DXVECTOR3    *pRotationCenter,
  _In_ const D3DXQUATERNION *pRotation,
  _In_ const D3DXVECTOR3    *pTranslation
);
```

Parameters

pOut [in]

Type: **D3DXMATRIX***

Pointer to the **D3DXMATRIX** that is the result of the operation.

Scaling [in]

Type: **FLOAT**

Scaling factor.

pRotationCenter [in]

Type: **const D3DXVECTOR3***

Pointer to a **D3DXVECTOR3**, a point identifying the center of rotation. If this argument is **NULL**, an identity M_{rc} matrix is applied to the formula in Remarks.

pRotation [in]

Type: **const D3DXQUATERNION***

Pointer to a **D3DXQUATERNION** that specifies the rotation. If this argument is **NULL**, an identity M_r matrix is applied to the formula in Remarks.

pTranslation [in]

Type: **const D3DXVECTOR3***

Pointer to a D3DXVECTOR3 structure representing the translation. If this argument is **NULL**, an identity M_t matrix is applied to the formula in Remarks.

Return value

Type: **D3DXMATRIX***

Pointer to a D3DXMATRIX structure that is an affine transformation matrix.

Remarks

This function calculates the affine transformation matrix with the following formula, with matrix concatenation evaluated in left-to-right order:

$M_{out} = M_s * (M_{rc})-1 * M_r * M_{rc} * M_t$

where:

M_{out} = output matrix (pOut)

M_s = scaling matrix (Scaling)

M_{rc} = center of rotation matrix (pRotationCenter)

M_r = rotation matrix (pRotation)

M_t = translation matrix (pTranslation)

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXMatrixAffineTransformation function can be used as a parameter for another function.

For 2D affine transformations, use D3DXMatrixAffineTransformation2D.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXMatrixAffineTransformation2D function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX utility library is deprecated. We recommend that you use **DirectXMath** instead.

Builds a 2D affine transformation matrix in the x-y plane. **NULL** arguments are treated as identity transformations.

Syntax

C++

```
D3DXMATRIX* D3DXMatrixAffineTransformation2D(
  _In_       D3DXMATRIX  *pOut,
  _In_       FLOAT       Scaling,
  _In_ const D3DXVECTOR2 *pRotationCenter,
  _In_       FLOAT       Rotation,
  _In_ const D3DXVECTOR2 *pTranslation
);
```

Parameters

pOut [in]

Type: **D3DXMATRIX***

Pointer to the **D3DXMATRIX** that is the result of the operation.

Scaling [in]

Type: **FLOAT**

Scaling factor.

pRotationCenter [in]

Type: **const D3DXVECTOR2***

Pointer to a **D3DXVECTOR2**, a point identifying the center of rotation. If this argument is **NULL**, an identity M_{rc} matrix is applied to the formula in Remarks.

Rotation [in]

Type: **FLOAT**

The angle of rotation.

pTranslation [in]

Type: **const D3DXVECTOR2***

Pointer to a **D3DXVECTOR2**, representing the translation. If this argument is **NULL**, an identity M_t matrix is applied to the formula in Remarks.

Return value

Type: **D3DXMATRIX***

Pointer to a D3DXMATRIX structure that is an affine transformation matrix.

Remarks

This function calculates the affine transformation matrix with the following formula, with matrix concatenation evaluated in left-to-right order:

$M_{out} = M_s * (M_{rc})-1 * M_r * M_{rc} * M_t$

where:

M_{out} = output matrix (pOut)

M_s = scaling matrix (Scaling)

M_{rc} = center of rotation matrix (pRotationCenter)

M_r = rotation matrix (Rotation)

M_t = translation matrix (pTranslation)

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXMatrixAffineTransformation2D function can be used as a parameter for another function.

For 3D affine transformations, use D3DXMatrixAffineTransformation.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXMatrixDecompose function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX utility library is deprecated. We recommend that you use **DirectXMath** instead.

Breaks down a general 3D transformation matrix into its scalar, rotational, and translational components.

Syntax

C++

```
HRESULT D3DXMatrixDecompose(
  _In_       D3DXVECTOR3    *pOutScale,
  _In_       D3DXQUATERNION *pOutRotation,
  _In_       D3DXVECTOR3    *pOutTranslation,
  _In_ const D3DXMATRIX     *pM
);
```

Parameters

pOutScale [in]

Type: **D3DXVECTOR3***

Pointer to the output **D3DXVECTOR3** that contains scaling factors applied along the x, y, and z-axes.

pOutRotation [in]

Type: **D3DXQUATERNION***

Pointer to the **D3DXQUATERNION** that describes the rotation.

pOutTranslation [in]

Type: **D3DXVECTOR3***

Pointer to the D3DXVECTOR3 vector that describes the translation.

pM [in]

Type: **const D3DXMATRIX***

Pointer to an input **D3DXMATRIX** matrix to decompose.

Return value

Type: **HRESULT**

If the function succeeds, the return value is S_OK. If the function fails, the return value can be the following: D3DERR_INVALIDCALL.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXMatrixDeterminant function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX utility library is deprecated. We recommend that you use **DirectXMath** instead.

Returns the determinant of a matrix.

Syntax

```C++
FLOAT D3DXMatrixDeterminant(
  _In_ const D3DXMATRIX *pM
);
```

Parameters

pM [in]

Type: **const** D3DXMATRIX*

Pointer to the source D3DXMATRIX structure.

Return value

Type: FLOAT

Returns the determinant of the matrix.

Requirements

Requirement	Value
Header	D3DX10Math.h

Requirement	Value
Library	D3DX10.lib

See also

Math Functions

D3DXMatrixInverse function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Calculates the inverse of a matrix.

Syntax

```C++
D3DXMATRIX* D3DXMatrixInverse(
  _Inout_       D3DXMATRIX *pOut,
  _Inout_       FLOAT      *pDeterminant,
  _In_    const D3DXMATRIX *pM
);
```

Parameters

pOut [in, out]

Type: **D3DXMATRIX***

Pointer to the **D3DXMATRIX** structure that is the result of the operation.

pDeterminant [in, out]

Type: **FLOAT***

Pointer to a FLOAT value containing the determinant of the matrix. If the determinant is not needed, set this parameter to **NULL**.

pM [in]

Type: **const D3DXMATRIX***

Pointer to the source D3DXMATRIX structure.

Return value

Type: **D3DXMATRIX***

Pointer to a D3DXMATRIX structure that is the inverse of the matrix. If matrix inversion fails, **NULL** is returned.

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXMatrixInverse function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXMatrixLookAtLH function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Builds a left-handed, look-at matrix.

Syntax

```C++
D3DXMATRIX* D3DXMatrixLookAtLH(
  _Inout_       D3DXMATRIX  *pOut,
  _In_    const D3DXVECTOR3 *pEye,
  _In_    const D3DXVECTOR3 *pAt,
  _In_    const D3DXVECTOR3 *pUp
);
```

Parameters

pOut [in, out]

Type: **D3DXMATRIX***

Pointer to the **D3DXMATRIX** structure that is the result of the operation.

pEye [in]

Type: **const D3DXVECTOR3***

Pointer to the **D3DXVECTOR3** that defines the eye point. This value is used in translation.

pAt [in]

Type: **const D3DXVECTOR3***

Pointer to the D3DXVECTOR3 structure that defines the camera look-at target.

pUp [in]

Type: **const** D3DXVECTOR3*

Pointer to the D3DXVECTOR3 structure that defines the current world's up, usually [0, 1, 0].

Return value

Type: D3DXMATRIX*

Pointer to a D3DXMATRIX structure that is a left-handed, look-at matrix.

Remarks

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXMatrixLookAtLH function can be used as a parameter for another function.

This function uses the following formula to compute the returned matrix.

```
zaxis = normal(At - Eye)
xaxis = normal(cross(Up, zaxis))
yaxis = cross(zaxis, xaxis)

 xaxis.x           yaxis.x           zaxis.x          0
 xaxis.y           yaxis.y           zaxis.y          0
 xaxis.z           yaxis.z           zaxis.z          0
-dot(xaxis, eye)  -dot(yaxis, eye)  -dot(zaxis, eye)  1
```

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXMatrixLookAtRH function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Builds a right-handed, look-at matrix.

Syntax

```C++
D3DXMATRIX* D3DXMatrixLookAtRH(
  _Inout_       D3DXMATRIX  *pOut,
  _In_    const D3DXVECTOR3 *pEye,
  _In_    const D3DXVECTOR3 *pAt,
  _In_    const D3DXVECTOR3 *pUp
);
```

Parameters

pOut [in, out]

Type: **D3DXMATRIX***

Pointer to the **D3DXMATRIX** structure that is the result of the operation.

pEye [in]

Type: **const D3DXVECTOR3***

Pointer to the **D3DXVECTOR3** that defines the eye point. This value is used in translation.

pAt [in]

Type: **const D3DXVECTOR3***

Pointer to the D3DXVECTOR3 structure that defines the camera look-at target.

pUp [in]

Type: **const** D3DXVECTOR3*

Pointer to the D3DXVECTOR3 structure that defines the current world's up, usually [0, 1, 0].

Return value

Type: D3DXMATRIX*

Pointer to a D3DXMATRIX structure that is a right-handed, look-at matrix.

Remarks

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXMatrixLookAtRH function can be used as a parameter for another function.

This function uses the following formula to compute the returned matrix.

```
zaxis = normal(Eye - At)
xaxis = normal(cross(Up, zaxis))
yaxis = cross(zaxis, xaxis)

 -xaxis.x           yaxis.x           -zaxis.x          0
 -xaxis.y           yaxis.y           -zaxis.y          0
 -xaxis.z           yaxis.z           -zaxis.z          0
dot(xaxis, eye)  -dot(yaxis, eye)  -dot(zaxis, eye)  1
```

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXMatrixMultiply function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Determines the product of two matrices.

Syntax

```C++
D3DXMATRIX* D3DXMatrixMultiply(
  _Inout_       D3DXMATRIX *pOut,
  _In_    const D3DXMATRIX *pM1,
  _In_    const D3DXMATRIX *pM2
);
```

Parameters

pOut [in, out]

Type: **D3DXMATRIX***

Pointer to the **D3DXMATRIX** structure that is the result of the operation.

pM1 [in]

Type: **const D3DXMATRIX***

Pointer to a source D3DXMATRIX structure (left hand side).

pM2 [in]

Type: **const D3DXMATRIX***

Pointer to a source D3DXMATRIX structure (right hand side).

Return value

Type: **D3DXMATRIX***

Pointer to a D3DXMATRIX structure that is the product of two matrices.

Remarks

The result represents the transformation M1 followed by the transformation M2 (Out = M1 * M2).

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXMatrixMultiply function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXMatrixMultiplyTranspose function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Calculates the transposed product of two matrices.

Syntax

```C++
D3DXMATRIX* D3DXMatrixMultiplyTranspose(
  _Inout_       D3DXMATRIX *pOut,
  _In_    const D3DXMATRIX *pM1,
  _In_    const D3DXMATRIX *pM2
);
```

Parameters

pOut [in, out]

Type: **D3DXMATRIX***

Pointer to the **D3DXMATRIX** structure that is the result of the operation.

pM1 [in]

Type: **const D3DXMATRIX***

Pointer to a source D3DXMATRIX structure (left hand side).

pM2 [in]

Type: **const D3DXMATRIX***

Pointer to a source D3DXMATRIX structure (right hand side).

Return value

Type: **D3DXMATRIX***

Pointer to a D3DXMATRIX structure that is the product of two matrices.

Remarks

The result is the transposed of the product of two transformation matrices, Out = T(M1*M2).

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXMatrixMultiplyTranspose function can be used as a parameter for another function.

This function is useful to set matrices as constants for vertex and pixel shaders.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXMatrixOrthoLH function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Builds a left-handed orthographic projection matrix.

Syntax

```C++
D3DXMATRIX* D3DXMatrixOrthoLH(
  _Inout_ D3DXMATRIX *pOut,
  _In_    FLOAT      w,
  _In_    FLOAT      h,
  _In_    FLOAT      zn,
  _In_    FLOAT      zf
);
```

Parameters

pOut [in, out]

Type: **D3DXMATRIX***

Pointer to the resulting **D3DXMATRIX**.

w [in]

Type: **FLOAT**

Width of the view volume.

h [in]

Type: **FLOAT**

Height of the view volume.

zn [in]

Type: **FLOAT**

Minimum z-value of the view volume which is referred to as z-near.

zf [in]

Type: **FLOAT**

Maximum z-value of the view volume which is referred to as z-far.

Return value

Type: **D3DXMATRIX***

Pointer to the resulting **D3DXMATRIX**.

Remarks

All the parameters of the D3DXMatrixOrthoLH function are distances in camera space. The parameters describe the dimensions of the view volume.

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXMatrixOrthoLH function can be used as a parameter for another function.

This function uses the following formula to compute the returned matrix.

```
2/w  0    0           0
0    2/h  0           0
0    0    1/(zf-zn)   0
0    0    zn/(zn-zf)  1
```

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXMatrixOrthoOffCenterLH function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Builds a customized, left-handed orthographic projection matrix.

Syntax

```C++
D3DXMATRIX* D3DXMatrixOrthoOffCenterLH(
  _Inout_ D3DXMATRIX *pOut,
  _In_    FLOAT      l,
  _In_    FLOAT      r,
  _In_    FLOAT      b,
  _In_    FLOAT      t,
  _In_    FLOAT      zn,
  _In_    FLOAT      zf
);
```

Parameters

pOut [in, out]

Type: **D3DXMATRIX***

Pointer to the resulting **D3DXMATRIX**.

l [in]

Type: **FLOAT**

Minimum x-value of view volume.

r [in]

Type: **FLOAT**

Maximum x-value of view volume.

b [in]

Type: **FLOAT**

Minimum y-value of view volume.

t [in]

Type: **FLOAT**

Maximum y-value of view volume.

zn [in]

Type: **FLOAT**

Minimum z-value of the view volume.

zf [in]

Type: **FLOAT**

Maximum z-value of the view volume.

Return value

Type: **D3DXMATRIX***

Pointer to the resulting **D3DXMATRIX**.

Remarks

The **D3DXMatrixOrthoLH** is a special case of the D3DXMatrixOrthoOffCenterLH function. To create the same projection using D3DXMatrixOrthoOffCenterLH, use the following values:

l = -w/2,

r = w/2,

b = -h/2, and

t = h/2.

All the parameters of the D3DXMatrixOrthoOffCenterLH function are distances in camera space. The parameters describe the dimensions of the view volume.

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXMatrixOrthoOffCenterLH function can be used as a parameter for another function.

This function uses the following formula to compute the returned matrix.

```
2/(r-l)      0            0           0
0            2/(t-b)      0           0
0            0            1/(zf-zn)   0
(l+r)/(l-r)  (t+b)/(b-t)  zn/(zn-zf)  1
```

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXMatrixOrthoOffCenterRH function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Builds a customized, right-handed orthographic projection matrix.

Syntax

```C++
D3DXMATRIX* D3DXMatrixOrthoOffCenterRH(
  _Inout_ D3DXMATRIX *pOut,
  _In_    FLOAT      l,
  _In_    FLOAT      r,
  _In_    FLOAT      b,
  _In_    FLOAT      t,
  _In_    FLOAT      zn,
  _In_    FLOAT      zf
);
```

Parameters

pOut [in, out]

Type: **D3DXMATRIX***

Pointer to the resulting **D3DXMATRIX**.

l [in]

Type: **FLOAT**

Minimum x-value of view volume.

r [in]

Type: **FLOAT**

Maximum x-value of view volume.

b [in]

Type: **FLOAT**

Minimum y-value of view volume.

t [in]

Type: **FLOAT**

Maximum y-value of view volume.

zn [in]

Type: **FLOAT**

Minimum z-value of the view volume.

zf [in]

Type: **FLOAT**

Maximum z-value of the view volume.

Return value

Type: **D3DXMATRIX***

Pointer to the resulting **D3DXMATRIX**.

Remarks

The **D3DXMatrixOrthoRH** function is a special case of the D3DXMatrixOrthoOffCenterRH function. To create the same projection using D3DXMatrixOrthoOffCenterRH, use the following values:

l = -w/2,

r = w/2,

b = -h/2, and

t = h/2.

All the parameters of the D3DXMatrixOrthoOffCenterRH function are distances in camera space. The parameters describe the dimensions of the view volume.

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXMatrixOrthoOffCenterRH function can be used as a parameter for another function.

This function uses the following formula to compute the returned matrix.

```
2/(r-l)      0            0           0
0            2/(t-b)      0           0
0            0            1/(zn-zf)   0
(l+r)/(l-r)  (t+b)/(b-t)  zn/(zn-zf)  1
```

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXMatrixOrthoRH function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Builds a right-handed orthographic projection matrix.

Syntax

```C++
D3DXMATRIX* D3DXMatrixOrthoRH(
  _Inout_ D3DXMATRIX *pOut,
  _In_    FLOAT      w,
  _In_    FLOAT      h,
  _In_    FLOAT      zn,
  _In_    FLOAT      zf
);
```

Parameters

pOut [in, out]

Type: **D3DXMATRIX***

Pointer to the resulting **D3DXMATRIX**.

w [in]

Type: **FLOAT**

Width of the view volume.

h [in]

Type: **FLOAT**

Height of the view volume.

zn [in]

Type: **FLOAT**

Minimum z-value of the view volume.

zf [in]

Type: **FLOAT**

Maximum z-value of the view volume.

Return value

Type: **D3DXMATRIX***

Pointer to the resulting **D3DXMATRIX**.

Remarks

All the parameters of the D3DXMatrixOrthoRH function are distances in camera space. The parameters describe the dimensions of the view volume.

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXMatrixOrthoRH function can be used as a parameter for another function.

This function uses the following formula to compute the returned matrix.

```
2/w  0    0           0
0    2/h  0           0
0    0    1/(zn-zf)   0
0    0    zn/(zn-zf)  1
```

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXMatrixPerspectiveFovLH function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Builds a left-handed perspective projection matrix based on a field of view.

Syntax

```C++
D3DXMATRIX* D3DXMatrixPerspectiveFovLH(
  _Inout_ D3DXMATRIX *pOut,
  _In_    FLOAT      fovy,
  _In_    FLOAT      Aspect,
  _In_    FLOAT      zn,
  _In_    FLOAT      zf
);
```

Parameters

pOut [in, out]

Type: **D3DXMATRIX***

Pointer to the **D3DXMATRIX** structure that is the result of the operation.

fovy [in]

Type: **FLOAT**

Field of view in the y direction, in radians.

Aspect [in]

Type: **FLOAT**

Aspect ratio, defined as view space width divided by height.

zn [in]

Type: **FLOAT**

Z-value of the near view-plane.

zf [in]

Type: **FLOAT**

Z-value of the far view-plane.

Return value

Type: **D3DXMATRIX***

Pointer to a D3DXMATRIX structure that is a left-handed perspective projection matrix.

Remarks

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXMatrixPerspectiveFovLH function can be used as a parameter for another function.

This function computes the returned matrix as shown:

```
xScale     0          0               0
0        yScale       0               0
0          0       zf/(zf-zn)         1
0          0       -zn*zf/(zf-zn)     0
where:
yScale = cot(fovY/2)

xScale = yScale / aspect ratio
```

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXMatrixPerspectiveFovRH function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Builds a right-handed perspective projection matrix based on a field of view.

Syntax

```C++
D3DXMATRIX* D3DXMatrixPerspectiveFovRH(
  _Inout_ D3DXMATRIX *pOut,
  _In_    FLOAT      fovy,
  _In_    FLOAT      Aspect,
  _In_    FLOAT      zn,
  _In_    FLOAT      zf
);
```

Parameters

pOut [in, out]

Type: **D3DXMATRIX***

Pointer to the **D3DXMATRIX** structure that is the result of the operation.

fovy [in]

Type: **FLOAT**

Field of view in the y direction, in radians.

Aspect [in]

Type: **FLOAT**

Aspect ratio, defined as view space width divided by height.

zn [in]

Type: **FLOAT**

Z-value of the near view-plane.

zf [in]

Type: **FLOAT**

Z-value of the far view-plane.

Return value

Type: **D3DXMATRIX***

Pointer to a D3DXMATRIX structure that is a right-handed perspective projection matrix.

Remarks

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXMatrixPerspectiveFovRH function can be used as a parameter for another function.

This function computes the returned matrix as shown.

```
xScale     0          0              0
0        yScale       0              0
0          0      zf/(zn-zf)        -1
0          0      zn*zf/(zn-zf)      0
where:
yScale = cot(fovY/2)

xScale = yScale / aspect ratio
```

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXMatrixPerspectiveLH function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Builds a left-handed perspective projection matrix

Syntax

```C++
D3DXMATRIX* D3DXMatrixPerspectiveLH(
  _Inout_ D3DXMATRIX *pOut,
  _In_    FLOAT      w,
  _In_    FLOAT      h,
  _In_    FLOAT      zn,
  _In_    FLOAT      zf
);
```

Parameters

pOut [in, out]

Type: **D3DXMATRIX***

Pointer to the **D3DXMATRIX** structure that is the result of the operation.

w [in]

Type: **FLOAT**

Width of the view volume at the near view-plane.

h [in]

Type: **FLOAT**

Height of the view volume at the near view-plane.

zn [in]

Type: **FLOAT**

Z-value of the near view-plane.

zf [in]

Type: **FLOAT**

Z-value of the far view-plane.

Return value

Type: **D3DXMATRIX***

Pointer to a D3DXMATRIX structure that is a left-handed perspective projection matrix.

Remarks

All the parameters of the D3DXMatrixPerspectiveLH function are distances in camera space. The parameters describe the dimensions of the view volume.

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXMatrixPerspectiveLH function can be used as a parameter for another function.

This function uses the following formula to compute the returned matrix.

```
2*zn/w  0       0              0
0       2*zn/h  0              0
0       0       zf/(zf-zn)     1
0       0       zn*zf/(zn-zf)  0
```

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXMatrixPerspectiveOffCenterLH function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Builds a customized, left-handed perspective projection matrix.

Syntax

```C++
D3DXMATRIX* D3DXMatrixPerspectiveOffCenterLH(
  _Inout_ D3DXMATRIX *pOut,
  _In_    FLOAT      l,
  _In_    FLOAT      r,
  _In_    FLOAT      b,
  _In_    FLOAT      t,
  _In_    FLOAT      zn,
  _In_    FLOAT      zf
);
```

Parameters

pOut [in, out]

Type: **D3DXMATRIX***

Pointer to the **D3DXMATRIX** structure that is the result of the operation.

l [in]

Type: **FLOAT**

Minimum x-value of the view volume.

r [in]

Type: **FLOAT**

Maximum x-value of the view volume.

b [in]

Type: **FLOAT**

Minimum y-value of the view volume.

t [in]

Type: **FLOAT**

Maximum y-value of the view volume.

zn [in]

Type: **FLOAT**

Minimum z-value of the view volume.

zf [in]

Type: **FLOAT**

Maximum z-value of the view volume.

Return value

Type: **D3DXMATRIX***

Pointer to a D3DXMATRIX structure that is a customized, left-handed perspective projection matrix.

Remarks

All the parameters of the D3DXMatrixPerspectiveOffCenterLH function are distances in camera space. The parameters describe the dimensions of the view volume.

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXMatrixPerspectiveOffCenterLH function can be used as a parameter for another function.

This function uses the following formula to compute the returned matrix.

```
2*zn/(r-l)   0            0              0
0            2*zn/(t-b)   0              0
(l+r)/(l-r)  (t+b)/(b-t)  zf/(zf-zn)     1
0            0            zn*zf/(zn-zf)  0
```

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXMatrixPerspectiveOffCenterRH function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Builds a customized, right-handed perspective projection matrix.

Syntax

```C++
D3DXMATRIX* D3DXMatrixPerspectiveOffCenterRH(
  _Inout_ D3DXMATRIX *pOut,
  _In_    FLOAT      l,
  _In_    FLOAT      r,
  _In_    FLOAT      b,
  _In_    FLOAT      t,
  _In_    FLOAT      zn,
  _In_    FLOAT      zf
);
```

Parameters

pOut [in, out]

Type: **D3DXMATRIX***

Pointer to the **D3DXMATRIX** structure that is the result of the operation.

l [in]

Type: **FLOAT**

Minimum x-value of the view volume.

r [in]

Type: **FLOAT**

Maximum x-value of the view volume.

b [in]

Type: **FLOAT**

Minimum y-value of the view volume.

t [in]

Type: **FLOAT**

Maximum y-value of the view volume.

zn [in]

Type: **FLOAT**

Minimum z-value of the view volume.

zf [in]

Type: **FLOAT**

Maximum z-value of the view volume.

Return value

Type: **D3DXMATRIX***

Pointer to a D3DXMATRIX structure that is a customized, right-handed perspective projection matrix.

Remarks

All the parameters of the D3DXMatrixPerspectiveOffCenterRH function are distances in camera space. The parameters describe the dimensions of the view volume.

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXMatrixPerspectiveOffCenterRH function can be used as a parameter for another function.

This function uses the following formula to compute the returned matrix.

```
2*zn/(r-l)   0            0                0
0            2*zn/(t-b)   0                0
(l+r)/(r-l)  (t+b)/(t-b)  zf/(zn-zf)      -1
0            0            zn*zf/(zn-zf)    0
```

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXMatrixPerspectiveRH function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Builds a right-handed perspective projection matrix.

Syntax

```C++
D3DXMATRIX* D3DXMatrixPerspectiveRH(
  _Inout_ D3DXMATRIX *pOut,
  _In_    FLOAT      w,
  _In_    FLOAT      h,
  _In_    FLOAT      zn,
  _In_    FLOAT      zf
);
```

Parameters

pOut [in, out]

Type: **D3DXMATRIX***

Pointer to the **D3DXMATRIX** structure that is the result of the operation.

w [in]

Type: **FLOAT**

Width of the view volume at the near view-plane.

h [in]

Type: **FLOAT**

Height of the view volume at the near view-plane.

zn [in]

Type: **FLOAT**

Z-value of the near view-plane.

zf [in]

Type: **FLOAT**

Z-value of the far view-plane.

Return value

Type: **D3DXMATRIX***

Pointer to a D3DXMATRIX structure that is a right-handed perspective projection matrix.

Remarks

All the parameters of the D3DXMatrixPerspectiveRH function are distances in camera space. The parameters describe the dimensions of the view volume.

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXMatrixPerspectiveRH function can be used as a parameter for another function.

This function uses the following formula to compute the returned matrix.

```
2*zn/w  0       0              0
0       2*zn/h  0              0
0       0       zf/(zn-zf)    -1
0       0       zn*zf/(zn-zf)  0
```

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXMatrixReflect function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Builds a matrix that reflects the coordinate system about a plane.

Syntax

```C++
D3DXMATRIX* D3DXMatrixReflect(
  _Inout_       D3DXMATRIX *pOut,
  _In_    const D3DXPLANE  *pPlane
);
```

Parameters

pOut [in, out]

Type: **D3DXMATRIX***

Pointer to the **D3DXMATRIX** structure that is the result of the operation.

pPlane [in]

Type: **const D3DXPLANE***

Pointer to the source **D3DXPLANE**.

Return value

Type: **D3DXMATRIX***

Pointer to a D3DXMATRIX structure that reflects the coordinate system about the source plane.

Remarks

This function normalizes the plane equation before it creates the reflected matrix.

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXMatrixReflect function can be used as a parameter for another function.

This function uses the following formula to compute the returned matrix.

```
P = normalize(Plane);

-2 * P.a * P.a + 1  -2 * P.b * P.a      -2 * P.c * P.a        0
-2 * P.a * P.b      -2 * P.b * P.b + 1  -2 * P.c * P.b        0
-2 * P.a * P.c      -2 * P.b * P.c      -2 * P.c * P.c + 1    0
-2 * P.a * P.d      -2 * P.b * P.d      -2 * P.c * P.d        1
```

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXMatrixRotationAxis function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Builds a matrix that rotates around an arbitrary axis.

Syntax

```C++
D3DXMATRIX* D3DXMatrixRotationAxis(
  _Inout_       D3DXMATRIX  *pOut,
  _In_    const D3DXVECTOR3 *pV,
  _In_          FLOAT       Angle
);
```

Parameters

pOut [in, out]

Type: **D3DXMATRIX***

Pointer to the **D3DXMATRIX** structure that is the result of the operation.

pV [in]

Type: **const D3DXVECTOR3***

Pointer to the arbitrary axis. See **D3DXVECTOR3**.

Angle [in]

Type: **FLOAT**

Angle of rotation in radians. Angles are measured clockwise when looking along the rotation axis toward the origin.

Return value

Type: **D3DXMATRIX***

Pointer to a D3DXMATRIX structure rotated around the specified axis.

Remarks

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXMatrixRotationAxis function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXMatrixRotationQuaternion function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Builds a rotation matrix from a quaternion.

Syntax

```
C++

D3DXMATRIX* D3DXMatrixRotationQuaternion(
  _Inout_       D3DXMATRIX     *pOut,
  _In_    const D3DXQUATERNION *pQ
);
```

Parameters

pOut [in, out]

Type: **D3DXMATRIX***

Pointer to the **D3DXMATRIX** structure that is the result of the operation.

pQ [in]

Type: **const D3DXQUATERNION***

Pointer to the source D3DXQUATERNION structure.

Return value

Type: **D3DXMATRIX***

Pointer to a D3DXMATRIX structure built from the source quaternion.

Remarks

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXMatrixRotationQuaternion function can be used as a parameter for another function.

For information about how to calculate quaternion values from a direction vector (x, y, z) and an angle of rotation, see **D3DXQUATERNION**.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXMatrixRotationX function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Builds a matrix that rotates around the x-axis.

Syntax

```C++
D3DXMATRIX* D3DXMatrixRotationX(
  _Inout_ D3DXMATRIX *pOut,
  _In_    FLOAT      Angle
);
```

Parameters

pOut [in, out]

Type: **D3DXMATRIX***

Pointer to the **D3DXMATRIX** structure that is the result of the operation.

Angle [in]

Type: **FLOAT**

Angle of rotation in radians. Angles are measured clockwise when looking along the rotation axis toward the origin.

Return value

Type: **D3DXMATRIX***

Pointer to a D3DXMATRIX structure rotated around the x-axis.

Remarks

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXMatrixRotationX function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXMatrixRotationY function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Builds a matrix that rotates around the y-axis.

Syntax

```C++
D3DXMATRIX* D3DXMatrixRotationY(
  _Inout_ D3DXMATRIX *pOut,
  _In_    FLOAT      Angle
);
```

Parameters

pOut [in, out]

Type: **D3DXMATRIX***

Pointer to the **D3DXMATRIX** structure that is the result of the operation.

Angle [in]

Type: **FLOAT**

Angle of rotation in radians. Angles are measured clockwise when looking along the rotation axis toward the origin.

Return value

Type: **D3DXMATRIX***

Pointer to a D3DXMATRIX structure rotated around the y-axis.

Remarks

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXMatrixRotationY function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXMatrixRotationYawPitchRoll function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Builds a matrix with a specified yaw, pitch, and roll.

Syntax

```
C++

D3DXMATRIX* D3DXMatrixRotationYawPitchRoll(
  _Inout_ D3DXMATRIX *pOut,
  _In_    FLOAT      Yaw,
  _In_    FLOAT      Pitch,
  _In_    FLOAT      Roll
);
```

Parameters

pOut [in, out]

Type: **D3DXMATRIX***

Pointer to the **D3DXMATRIX** structure that is the result of the operation.

Yaw [in]

Type: **FLOAT**

Yaw around the y-axis, in radians.

Pitch [in]

Type: **FLOAT**

Pitch around the x-axis, in radians.

Roll [in]

Type: **FLOAT**

Roll around the z-axis, in radians.

Return value

Type: **D3DXMATRIX***

Pointer to a D3DXMATRIX structure with the specified yaw, pitch, and roll.

Remarks

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXMatrixRotationYawPitchRoll function can be used as a parameter for another function.

The order of transformations is roll first, then pitch, then yaw. Relative to the object's local coordinate axis, this is equivalent to rotation around the z-axis, followed by rotation around the x-axis, followed by rotation around the y-axis, as shown in the following illustration.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXMatrixRotationZ function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Builds a matrix that rotates around the z-axis.

Syntax

```C++
D3DXMATRIX* D3DXMatrixRotationZ(
  _Inout_ D3DXMATRIX *pOut,
  _In_    FLOAT      Angle
);
```

Parameters

pOut [in, out]

Type: **D3DXMATRIX***

Pointer to the **D3DXMATRIX** structure that is the result of the operation.

Angle [in]

Type: **FLOAT**

Angle of rotation, in radians. Angles are measured clockwise when viewed from the rotation axis (positive side) toward the origin.

Return value

Type: **D3DXMATRIX***

Pointer to a D3DXMATRIX structure rotated around the z-axis.

Remarks

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXMatrixRotationZ function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXMatrixScaling function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Builds a matrix that scales along the x-axis, the y-axis, and the z-axis.

Syntax

```C++
D3DXMATRIX* D3DXMatrixScaling(
  _Inout_ D3DXMATRIX *pOut,
  _In_    FLOAT      sx,
  _In_    FLOAT      sy,
  _In_    FLOAT      sz
);
```

Parameters

pOut [in, out]

Type: **D3DXMATRIX***

Pointer to the **D3DXMATRIX** structure that is the result of the operation.

sx [in]

Type: **FLOAT**

Scaling factor that is applied along the x-axis.

sy [in]

Type: **FLOAT**

Scaling factor that is applied along the y-axis.

sz [in]

Type: **FLOAT**

Scaling factor that is applied along the z-axis.

Return value

Type: **D3DXMATRIX***

Pointer to the scaling transformation D3DXMATRIX.

Remarks

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXMatrixScaling function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXMatrixShadow function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Builds a matrix that flattens geometry into a plane.

Syntax

```C++
D3DXMATRIX* D3DXMatrixShadow(
  _Inout_       D3DXMATRIX  *pOut,
  _In_    const D3DXVECTOR4 *pLight,
  _In_    const D3DXPLANE   *pPlane
);
```

Parameters

pOut [in, out]

Type: **D3DXMATRIX***

Pointer to the **D3DXMATRIX** structure that is the result of the operation.

pLight [in]

Type: **const D3DXVECTOR4***

Pointer to a **D3DXVECTOR4** describing the light's position.

pPlane [in]

Type: **const D3DXPLANE***

Pointer to the source **D3DXPLANE**.

Return value

Type: D3DXMATRIX*

Pointer to a D3DXMATRIX structure that flattens geometry into a plane.

Remarks

The **D3DXMatrixShadow** function flattens geometry into a plane, as if casting a shadow from a light.

The return value for this function is the same value returned in the pOut parameter. In this way, the **D3DXMatrixShadow** function can be used as a parameter for another function.

This function uses the following formula to compute the returned matrix.

```
P = normalize(Plane);
L = Light;
d = dot(P, L)

P.a * L.x + d  P.a * L.y      P.a * L.z      P.a * L.w
P.b * L.x      P.b * L.y + d  P.b * L.z      P.b * L.w
P.c * L.x      P.c * L.y      P.c * L.z + d  P.c * L.w
P.d * L.x      P.d * L.y      P.d * L.z      P.d * L.w + d
```

If the light's w-component is 0, the ray from the origin to the light represents a directional light. If it is 1, the light is a point light.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXMatrixTransformation function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Builds a transformation matrix. **NULL** arguments are treated as identity transformations.

Syntax

```C++
D3DXMATRIX* D3DXMatrixTransformation(
  _Inout_       D3DXMATRIX     *pOut,
  _In_    const D3DXVECTOR3    *pScalingCenter,
  _In_    const D3DXQUATERNION *pScalingRotation,
  _In_    const D3DXVECTOR3    *pScaling,
  _In_    const D3DXVECTOR3    *pRotationCenter,
  _In_    const D3DXQUATERNION *pRotation,
  _In_    const D3DXVECTOR3    *pTranslation
);
```

Parameters

pOut [in, out]

Type: **D3DXMATRIX***

Pointer to the **D3DXMATRIX** structure that is the result of the operation.

pScalingCenter [in]

Type: **const D3DXVECTOR3***

Pointer to a **D3DXVECTOR3**, identifying the scaling center point. If this argument is **NULL**, an identity M_{sc} matrix is applied to the formula in Remarks.

pScalingRotation [in]

Type: **const** D3DXQUATERNION*

Pointer to a D3DXQUATERNION that specifies the scaling rotation. If this argument is **NULL**, an identity M_{sr} matrix is applied to the formula in Remarks.

pScaling [in]

Type: **const** D3DXVECTOR3*

Pointer to a D3DXVECTOR3 structure, the scaling vector. If this argument is **NULL**, an identity M_s matrix is applied to the formula in Remarks.

pRotationCenter [in]

Type: **const** D3DXVECTOR3*

Pointer to a D3DXVECTOR3 structure, a point that identifies the center of rotation. If this argument is **NULL**, an identity M_{rc} matrix is applied to the formula in Remarks.

pRotation [in]

Type: **const** D3DXQUATERNION*

Pointer to a D3DXQUATERNION structure that specifies the rotation. If this argument is **NULL**, an identity M_r matrix is applied to the formula in Remarks.

pTranslation [in]

Type: **const** D3DXVECTOR3*

Pointer to a D3DXVECTOR3 structure, representing the translation. If this argument is **NULL**, an identity M_t matrix is applied to the formula in Remarks.

Return value

Type: D3DXMATRIX*

Pointer to a D3DXMATRIX structure that is the transformation matrix.

Remarks

This function calculates the transformation matrix with the following formula, with matrix concatenation evaluated in left-to-right order:

$M_{out} = (M_{sc})^{-1} * (M_{sr})^{-1} * M_s * M_{sr} * M_{sc} * (M_{rc})^{-1} * M_r * M_{rc} * Mt$

where:

M_{out} = output matrix (pOut)

M_{sc} = scaling center matrix (pScalingCenter)

M_{sr} = scaling rotation matrix (pScalingRotation)

M_{s} = scaling matrix (pScaling)

M_{rc} = center of rotation matrix (pRotationCenter)

M_{r} = rotation matrix (pRotation)

M_{t} = translation matrix (pTranslation)

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXMatrixTransformation function can be used as a parameter for another function.

For 2D transformations, use **D3DXMatrixTransformation2D**.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXMatrixTransformation2D function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Builds a 2D transformation matrix that represents transformations in the xy plane. **NULL** arguments are treated as identity transformations.

Syntax

C++

```
D3DXMATRIX* D3DXMatrixTransformation2D(
  _Inout_       D3DXMATRIX  *pOut,
  _In_    const D3DXVECTOR2 *pScalingCenter,
  _In_          FLOAT       ScalingRotation,
  _In_    const D3DXVECTOR2 *pScaling,
  _In_    const D3DXVECTOR2 *pRotationCenter,
  _In_          FLOAT       Rotation,
  _In_    const D3DXVECTOR2 *pTranslation
);
```

Parameters

pOut [in, out]

Type: **D3DXMATRIX***

Pointer to the **D3DXMATRIX** structure that contains the result of the transformations.

pScalingCenter [in]

Type: **const D3DXVECTOR2***

Pointer to a **D3DXVECTOR2**, a point identifying the scaling center. If this argument is **NULL**, an identity M_{sc} matrix is applied to the formula in Remarks.

ScalingRotation [in]

Type: **FLOAT**

Pointer to the scaling rotation factor.

pScaling [in]

Type: **const D3DXVECTOR2***

Pointer to a D3DXVECTOR2 structure, a point identifying the scale. If this argument is **NULL**, an identity M_s matrix is applied to the formula in Remarks.

pRotationCenter [in]

Type: **const D3DXVECTOR2***

Pointer to a D3DXVECTOR2 structure, a point identifying the rotation center. If this argument is **NULL**, an identity M_{rc} matrix is applied to the formula in Remarks.

Rotation [in]

Type: **FLOAT**

The angle of rotation in radians.

pTranslation [in]

Type: **const D3DXVECTOR2***

Pointer to a D3DXVECTOR2 structure, identifying the translation. If this argument is **NULL**, an identity M_t matrix is applied to the formula in Remarks.

Return value

Type: **D3DXMATRIX***

Pointer to a D3DXMATRIX structure that contains the transformation matrix.

Remarks

This function calculates the transformation matrix with the following formula, with matrix concatenation evaluated in left-to-right order:

$M_{out} = (M_{sc})^{-1} * (M_{sr})^{-1} * M_s * M_{sr} * M_{sc} * (M_{rc})^{-1} * M_r * M_{rc} * M_t$

where:

M_{out} = output matrix (pOut)

M_{sc} = scaling center matrix (pScalingCenter)

M_{sr} = scaling rotation matrix (pScalingRotation)

M_{s} = scaling matrix (pScaling)

M_{rc} = center of rotation matrix (pRotationCenter)

M_{r} = rotation matrix (Rotation)

M_{t} = translation matrix (pTranslation)

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXMatrixTransformation2D function can be used as a parameter for another function.

For 3D transformations, use **D3DXMatrixTransformation**.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXMatrixTranslation function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Create a translation matrix.

Syntax

```C++
D3DXMATRIX* D3DXMatrixTranslation(
  _Inout_ D3DXMATRIX *pOut,
  _In_    FLOAT      x,
  _In_    FLOAT      y,
  _In_    FLOAT      z
);
```

Parameters

pOut [in, out]

Type: **D3DXMATRIX***

The matrix that will become a translation matrix. See **D3DXMATRIX**.

x [in]

Type: **FLOAT**

The x-component of the translation.

y [in]

Type: **FLOAT**

The y-component of the translation.

z [in]

Type: **FLOAT**

The z-component of the translation.

Return value

Type: **D3DXMATRIX***

The translation matrix.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXMatrixTranspose function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Returns the matrix transpose of a matrix.

Syntax

```C++
D3DXMATRIX* D3DXMatrixTranspose(
  _Inout_       D3DXMATRIX *pOut,
  _In_    const D3DXMATRIX *pM
);
```

Parameters

pOut [in, out]

Type: **D3DXMATRIX***

Pointer to the **D3DXMATRIX** structure that is the result of the operation.

pM [in]

Type: **const D3DXMATRIX***

Pointer to the source D3DXMATRIX structure.

Return value

Type: **D3DXMATRIX***

Pointer to the D3DXMATRIX structure that is the matrix transpose of the matrix.

Remarks

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXMatrixTranspose function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXPlaneFromPointNormal function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Constructs a plane from a point and a normal.

Syntax

```C++
D3DXPLANE* D3DXPlaneFromPointNormal(
  _Inout_       D3DXPLANE   *pOut,
  _In_    const D3DXVECTOR3 *pPoint,
  _In_    const D3DXVECTOR3 *pNormal
);
```

Parameters

pOut [in, out]

Type: **D3DXPLANE***

Pointer to the **D3DXPLANE** that is the result of the operation.

pPoint [in]

Type: **const D3DXVECTOR3***

Pointer to a **D3DXVECTOR3**, defining the point used to construct the plane.

pNormal [in]

Type: **const D3DXVECTOR3***

Pointer to a D3DXVECTOR3 structure, defining the normal used to construct the plane.

Return value

Type: **D3DXPLANE***

Pointer to the D3DXPLANE structure constructed from the point and the normal.

Remarks

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXPlaneFromPointNormal function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXPlaneFromPoints function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Constructs a plane from three points.

Syntax

```C++
D3DXPLANE* D3DXPlaneFromPoints(
  _Inout_       D3DXPLANE   *pOut,
  _In_    const D3DXVECTOR3 *pV1,
  _In_    const D3DXVECTOR3 *pV2,
  _In_    const D3DXVECTOR3 *pV3
);
```

Parameters

pOut [in, out]

Type: **D3DXPLANE***

Pointer to the **D3DXPLANE** that is the result of the operation.

pV1 [in]

Type: **const D3DXVECTOR3***

Pointer to a **D3DXVECTOR3**, defining one of the points used to construct the plane.

pV2 [in]

Type: **const D3DXVECTOR3***

Pointer to a D3DXVECTOR3 structure, defining one of the points used to construct the plane.

pV3 [in]

Type: **const** D3DXVECTOR3*

Pointer to a D3DXVECTOR3 structure, defining one of the points used to construct the plane.

Return value

Type: D3DXPLANE*

Pointer to the D3DXPLANE structure constructed from the given points.

Remarks

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXPlaneFromPoints function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXPlaneIntersectLine function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Finds the intersection between a plane and a line.

Syntax

```
C++

D3DXVECTOR3* D3DXPlaneIntersectLine(
  _Inout_       D3DXVECTOR3 *pOut,
  _In_    const D3DXPLANE   *pP,
  _In_    const D3DXVECTOR3 *pV1,
  _In_    const D3DXVECTOR3 *pV2
);
```

Parameters

pOut [in, out]

Type: **D3DXVECTOR3***

Pointer to a **D3DXVECTOR3**, identifying the intersection between the specified plane and line.

pP [in]

Type: **const D3DXPLANE***

Pointer to the source **D3DXPLANE**.

pV1 [in]

Type: **const D3DXVECTOR3***

Pointer to a source D3DXVECTOR3 structure, defining a line starting point.

pV2 [in]

Type: **const** D3DXVECTOR3*

Pointer to a source D3DXVECTOR3 structure, defining a line ending point.

Return value

Type: D3DXVECTOR3*

Pointer to a D3DXVECTOR3 structure that is the intersection between the specified plane and line.

Remarks

If the line is parallel to the plane, **NULL** is returned.

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXPlaneIntersectLine function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXPlaneNormalize function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Normalizes the plane coefficients so that the plane normal has unit length.

Syntax

```C++
D3DXPLANE* D3DXPlaneNormalize(
  _Inout_       D3DXPLANE *pOut,
  _In_    const D3DXPLANE *pP
);
```

Parameters

pOut [in, out]

Type: **D3DXPLANE***

Pointer to the **D3DXPLANE** that is the result of the operation.

pP [in]

Type: **const D3DXPLANE***

Pointer to the source D3DXPLANE structure.

Return value

Type: **D3DXPLANE***

Pointer to a D3DXPLANE structure that represents the normal of the plane.

Remarks

This function normalizes a plane so that |a,b,c| == 1.

The return value for this function is the same value returned in the pOut parameter. In this way, this function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXPlaneTransform function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Transforms a plane by a matrix. The input matrix is the inverse transpose of the actual transformation.

Syntax

C++

```
D3DXPLANE* D3DXPlaneTransform(
  _Inout_       D3DXPLANE  *pOut,
  _In_    const D3DXPLANE  *pP,
  _In_    const D3DXMATRIX *pM
);
```

Parameters

pOut [in, out]

Type: **D3DXPLANE***

Pointer to the **D3DXPLANE** that contains the resulting transformed plane. See example.

pP [in]

Type: **const D3DXPLANE***

Pointer to the input D3DXPLANE structure, which contains the plane that will be transformed. The vector (a,b,c) that describes the plane must be normalized before this function is called. See example.

pM [in]

Type: **const D3DXMATRIX***

Pointer to the source D3DXMATRIX structure, which contains the transformation values. This matrix must contain the inverse transpose of the transformation values.

Return value

Type: D3DXPLANE*

Pointer to a D3DXPLANE structure, representing the transformed plane. This is the same value returned in the pOut parameter so that this function can be used as a parameter for another function.

Remarks

Examples

This example transforms a plane by applying a non-uniform scale.

```
D3DXPLANE   planeNew;
D3DXPLANE   plane(0,1,1,0);
D3DXPlaneNormalize(&plane, &plane);

D3DXMATRIX  matrix;
D3DXMatrixScaling(&matrix, 1.0f,2.0f,3.0f);
D3DXMatrixInverse(&matrix, NULL, &matrix);
D3DXMatrixTranspose(&matrix, &matrix);
D3DXPlaneTransform(&planeNew, &plane, &matrix);
```

A plane is described by ax + by + cz + dw = 0. The first plane is created with (a,b,c,d) = (0,1,1,0), which is a plane described by y + z = 0. After scaling, the new plane contains (a,b,c,d) = (0, 0.353f, 0.235f, 0), which shows the new plane to be described by 0.353y + 0.235z = 0.

The parameter pM contains the inverse transpose of the transformation matrix. The inverse transpose is required by this method so that the normal vector of the transformed plane can be correctly transformed as well.

Requirements

Requirement	Value

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXPlaneTransformArray function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Transforms an array of planes by a matrix. The vectors that describe each plane must be normalized.

Syntax

```C++
D3DXPLANE* D3DXPlaneTransformArray(
  _Inout_       D3DXPLANE  *pOut,
  _In_          UINT       OutStride,
  _In_    const D3DXPLANE  *pP,
  _In_          UINT       PStride,
  _In_    const D3DXMATRIX *pM,
  _In_          UINT       n
);
```

Parameters

pOut [in, out]

Type: **D3DXPLANE***

Pointer to the **D3DXPLANE** structure that contains the resulting transformed plane. See Example.

OutStride [in]

Type: **UINT**

The stride of each transformed plane.

pP [in]

Type: **const** D3DXPLANE*

Pointer to the input D3DXPLANE structure, which contains the array of planes to transform. The vector (a, b, c) that describes the plane must be normalized before this function is called. See Example.

PStride [in]

Type: UINT

The stride of each non-transformed plane.

pM [in]

Type: **const** D3DXMATRIX*

Pointer to the source D3DXMATRIX structure, which contains the inverse transpose of the transformation values.

n [in]

Type: UINT

The number of planes to transform.

Return value

Type: D3DXPLANE*

Pointer to a D3DXPLANE structure, representing the transformed plane. This is the same value returned in the pOut parameter so that this function can be used as a parameter for another function.

Remarks

This example transforms one plane by applying a non-uniform scale.

```
#define ARRAYSIZE 4
D3DXPLANE planeNew[ARRAYSIZE];
D3DXPLANE plane[ARRAYSIZE];

for(int i = 0; i < ARRAYSIZE; i++)
{
    plane = D3DXPLANE( 0.0f, 1.0f, 1.0f, 0.0f );
```

```
    D3DXPlaneNormalize( &plane[i], &plane[i] );
}

D3DXMATRIX  matrix;
D3DXMatrixScaling( &matrix, 1.0f, 2.0f, 3.0f );
D3DXMatrixInverse( &matrix, NULL, &matrix );
D3DXMatrixTranspose( &matrix, &matrix );
D3DXPlaneTransformArray( &planeNew, sizeof (D3DXPLANE), &plane,
                         sizeof (D3DXPLANE), &matrix, ARRAYSIZE );
```

A plane is described by ax + by + cz + dw = 0. The first plane is created with (a,b,c,d) = (0,1,1,0), which is a plane described by y + z = 0. After scaling, the new plane contains (a,b,c,d) = (0, 0.353f, 0.235f, 0), which shows the new plane to be described by 0.353y + 0.235z = 0.

The parameter pM, contains the inverse transpose of the transformation matrix. The inverse transpose is required by this method so that the normal vector of the transformed plane can be correctly transformed as well.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXQuaternionBaryCentric function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Returns a quaternion in barycentric coordinates.

Syntax

```C++
D3DXQUATERNION* D3DXQuaternionBaryCentric(
  _Inout_       D3DXQUATERNION *pOut,
  _In_    const D3DXQUATERNION *pQ1,
  _In_    const D3DXQUATERNION *pQ2,
  _In_    const D3DXQUATERNION *pQ3,
  _In_          FLOAT          f,
  _In_          FLOAT          g
);
```

Parameters

pOut [in, out]

Type: **D3DXQUATERNION***

Pointer to the **D3DXQUATERNION** that is the result of the operation.

pQ1 [in]

Type: **const D3DXQUATERNION***

Pointer to a source D3DXQUATERNION structure.

pQ2 [in]

Type: **const D3DXQUATERNION***

Pointer to a source D3DXQUATERNION structure.

pQ3 [in]

Type: **const** **D3DXQUATERNION***

Pointer to a source D3DXQUATERNION structure.

f [in]

Type: **FLOAT**

Weighting factor. See Remarks.

g [in]

Type: **FLOAT**

Weighting factor. See Remarks.

Return value

Type: **D3DXQUATERNION***

Pointer to a D3DXQUATERNION structure in Barycentric coordinates.

Remarks

To compute the barycentric coordinates, the D3DXQuaternionBaryCentric function implements the following series of spherical linear interpolation operations:

```
Slerp(Slerp(Q1, Q2, f+g), Slerp(Q1, Q3, f+g), g/(f+g))
```

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXQuaternionBaryCentric function can be used as a parameter for another function.

Use **D3DXQuaternionNormalize** for any quaternion input that is not already normalized.

Barycentric coordinates define a point inside a triangle in terms of the triangle's vertices. For a more in-depth description of barycentric coordinates, see Mathworld's Barycentric Coordinates Description.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXQuaternionExp function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Calculates the exponential.

Syntax

```C++
D3DXQUATERNION* D3DXQuaternionExp(
  _Inout_       D3DXQUATERNION *pOut,
  _In_    const D3DXQUATERNION *pQ
);
```

Parameters

pOut [in, out]

Type: **D3DXQUATERNION***

Pointer to the **D3DXQUATERNION** that is the result of the operation.

pQ [in]

Type: **const D3DXQUATERNION***

Pointer to the source D3DXQUATERNION structure.

Return value

Type: **D3DXQUATERNION***

Pointer to a D3DXQUATERNION structure that is the exponential.

Remarks

This method converts a pure quaternion to a unit quaternion. D3DXQuaternionExp expects a pure quaternion, where w is ignored in the calculation (w == 0).

```
Given a pure quaternion defined by:
q = (0, theta * v);

This method calculates the exponential result.
exp(Q) = (cos(theta), sin(theta) * v)
```

where v is the vector portion of a quaternion.

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXQuaternionExp function can be used as a parameter for another function.

The **D3DXQuaternionSquadSetup** method can also be used to set up the control points of a quaternion.

Use **D3DXQuaternionNormalize** for any quaternion input that is not already normalized.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXQuaternionInverse function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Conjugates and renormalizes a quaternion.

Syntax

```C++
D3DXQUATERNION* D3DXQuaternionInverse(
  _Inout_       D3DXQUATERNION *pOut,
  _In_    const D3DXQUATERNION *pQ
);
```

Parameters

pOut [in, out]

Type: **D3DXQUATERNION***

Pointer to the **D3DXQUATERNION** that is the result of the operation.

pQ [in]

Type: **const D3DXQUATERNION***

Pointer to the source D3DXQUATERNION structure.

Return value

Type: **D3DXQUATERNION***

Pointer to a D3DXQUATERNION structure that is the inverse quaternion of the quaternion.

Remarks

```
A unit quaternion, is defined by:
Q == (cos(theta), sin(theta) * v) where |v| = 1
The natural logarithm of Q is, ln(Q) = (0, theta * v)
```

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXQuaternionInverse function can be used as a parameter for another function.

Use **D3DXQuaternionNormalize** for any quaternion input that is not already normalized.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXQuaternionLn function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Calculates the natural logarithm.

Syntax

```C++
D3DXQUATERNION* D3DXQuaternionLn(
  _Inout_       D3DXQUATERNION *pOut,
  _In_    const D3DXQUATERNION *pQ
);
```

Parameters

pOut [in, out]

Type: **D3DXQUATERNION***

Pointer to the **D3DXQUATERNION** that is the result of the operation.

pQ [in]

Type: **const D3DXQUATERNION***

Pointer to the source D3DXQUATERNION structure.

Return value

Type: **D3DXQUATERNION***

Pointer to a D3DXQUATERNION structure that is the natural logarithm.

Remarks

The D3DXQuaternionLn function works only for unit quaternions.

```
A unit quaternion, is defined by:
Q == (cos(theta), sin(theta) * v) where |v| = 1
The natural logarithm of Q is, ln(Q) = (0, theta * v)
```

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXQuaternionLn function can be used as a parameter for another function.

Use **D3DXQuaternionNormalize** for any quaternion input that is not already normalized.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXQuaternionMultiply function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Multiplies two quaternions.

Syntax

```
C++

D3DXQUATERNION* D3DXQuaternionMultiply(
  _Inout_       D3DXQUATERNION *pOut,
  _In_    const D3DXQUATERNION *pQ1,
  _In_    const D3DXQUATERNION *pQ2
);
```

Parameters

pOut [in, out]

Type: **D3DXQUATERNION***

Pointer to the **D3DXQUATERNION** that is the result of the operation.

pQ1 [in]

Type: **const D3DXQUATERNION***

Pointer to a source **D3DXQUATERNION** structure.

pQ2 [in]

Type: **const D3DXQUATERNION***

Pointer to a source **D3DXQUATERNION** structure.

Return value

Type: D3DXQUATERNION*

Pointer to a D3DXQUATERNION structure that is the product of two quaternions.

Remarks

The result represents the rotation Q1 followed by the rotation Q2 (Out = Q2 * Q1). This is done so that **D3DXQuaternionMultiply** maintain the same semantics as D3DXMatrixMultiply because unit quaternions can be considered as another way to represent rotation matrices.

Transformations are concatenated in the same order for both the **D3DXQuaternionMultiply** and D3DXMatrixMultiply Functions. For example, assuming mX and mY represent the same rotations as qX and qY, both m and q will represent the same rotations.

```
D3DXMatrixMultiply(&m, &mX, &mY);
D3DXQuaternionMultiply(&q, &qX, &qY);
```

The multiplication of quaternions is not commutative.

The return value for this function is the same value returned in the pOut parameter. In this way, the **D3DXQuaternionMultiply** function can be used as a parameter for another function.

Use D3DXQuaternionNormalize for any quaternion input that is not already normalized.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXQuaternionNormalize function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Computes a unit length quaternion.

Syntax

```C++
D3DXQUATERNION* D3DXQuaternionNormalize(
  _Inout_       D3DXQUATERNION *pOut,
  _In_    const D3DXQUATERNION *pQ
);
```

Parameters

pOut [in, out]

Type: **D3DXQUATERNION***

Pointer to the **D3DXQUATERNION** that is the result of the operation.

pQ [in]

Type: **const D3DXQUATERNION***

Pointer to the source D3DXQUATERNION structure.

Return value

Type: **D3DXQUATERNION***

Pointer to a D3DXQUATERNION structure that is the normal of the quaternion.

Remarks

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXQuaternionNormalize function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXQuaternionRotationAxis function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Rotates a quaternion about an arbitrary axis.

Syntax

```C++
D3DXQUATERNION* D3DXQuaternionRotationAxis(
  _Inout_       D3DXQUATERNION *pOut,
  _In_    const D3DXVECTOR3    *pV,
  _In_          FLOAT          Angle
);
```

Parameters

pOut [in, out]

Type: **D3DXQUATERNION***

Pointer to the **D3DXQUATERNION** that is the result of the operation.

pV [in]

Type: **const D3DXVECTOR3***

Pointer to the **D3DXVECTOR3** that identifies the axis about which to rotate the quaternion.

Angle [in]

Type: **FLOAT**

Angle of rotation, in radians. Angles are measured clockwise when looking along the rotation axis toward the origin.

Return value

Type: **D3DXQUATERNION***

Pointer to a D3DXQUATERNION structure rotated around the specified axis.

Remarks

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXQuaternionRotationAxis function can be used as a parameter for another function.

Use **D3DXQuaternionNormalize** for any quaternion input that is not already normalized.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXQuaternionRotationMatrix function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Builds a quaternion from a rotation matrix.

Syntax

```C++
D3DXQUATERNION* D3DXQuaternionRotationMatrix(
  _Inout_       D3DXQUATERNION *pOut,
  _In_    const D3DXMATRIX     *pM
);
```

Parameters

pOut [in, out]

Type: **D3DXQUATERNION***

Pointer to the **D3DXQUATERNION** that is the result of the operation.

pM [in]

Type: **const D3DXMATRIX***

Pointer to the source **D3DXMATRIX** structure.

Return value

Type: **D3DXQUATERNION***

Pointer to the D3DXQUATERNION structure built from a rotation matrix.

Remarks

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXQuaternionRotationMatrix function can be used as a parameter for another function.

Use **D3DXQuaternionNormalize** for any quaternion input that is not already normalized.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXQuaternionRotationYawPitchRoll function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Builds a quaternion with the given yaw, pitch, and roll.

Syntax

```
C++

D3DXQUATERNION* D3DXQuaternionRotationYawPitchRoll(
  _Inout_ D3DXQUATERNION *pOut,
  _In_    FLOAT          Yaw,
  _In_    FLOAT          Pitch,
  _In_    FLOAT          Roll
);
```

Parameters

pOut [in, out]

Type: **D3DXQUATERNION***

Pointer to the **D3DXQUATERNION** that is the result of the operation.

Yaw [in]

Type: **FLOAT**

Yaw around the y-axis, in radians.

Pitch [in]

Type: **FLOAT**

Pitch around the x-axis, in radians.

Roll [in]

Type: **FLOAT**

Roll around the z-axis, in radians.

Return value

Type: **D3DXQUATERNION***

Pointer to a D3DXQUATERNION structure with the specified yaw, pitch, and roll.

Remarks

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXQuaternionRotationYawPitchRoll function can be used as a parameter for another function.

Use **D3DXQuaternionNormalize** for any quaternion input that is not already normalized.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXQuaternionSlerp function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Interpolates between two quaternions, using spherical linear interpolation.

Syntax

```C++
D3DXQUATERNION* D3DXQuaternionSlerp(
  _Inout_       D3DXQUATERNION *pOut,
  _In_    const D3DXQUATERNION *pQ1,
  _In_    const D3DXQUATERNION *pQ2,
  _In_          FLOAT          t
);
```

Parameters

pOut [in, out]

Type: **D3DXQUATERNION***

Pointer to the **D3DXQUATERNION** that is the result of the operation.

pQ1 [in]

Type: **const D3DXQUATERNION***

Pointer to a source D3DXQUATERNION structure.

pQ2 [in]

Type: **const D3DXQUATERNION***

Pointer to a source D3DXQUATERNION structure.

t [in]

Type: **FLOAT**

Parameter that indicates how far to interpolate between the quaternions.

Return value

Type: **D3DXQUATERNION***

Pointer to a D3DXQUATERNION structure that is the result of the interpolation.

Remarks

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXQuaternionSlerp function can be used as a parameter for another function.

Use **D3DXQuaternionNormalize** for any quaternion input that is not already normalized.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXQuaternionSquad function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Interpolates between quaternions, using spherical quadrangle interpolation.

Syntax

```C++
D3DXQUATERNION* D3DXQuaternionSquad(
  _Inout_       D3DXQUATERNION *pOut,
  _In_    const D3DXQUATERNION *pQ1,
  _In_    const D3DXQUATERNION *pA,
  _In_    const D3DXQUATERNION *pB,
  _In_    const D3DXQUATERNION *pC,
  _In_          FLOAT          t
);
```

Parameters

pOut [in, out]

Type: **D3DXQUATERNION***

Pointer to the **D3DXQUATERNION** that is the result of the operation.

pQ1 [in]

Type: **const D3DXQUATERNION***

Pointer to a source D3DXQUATERNION structure.

pA [in]

Type: **const D3DXQUATERNION***

Pointer to a source D3DXQUATERNION structure.

pB [in]

Type: **const** D3DXQUATERNION*

Pointer to a source D3DXQUATERNION structure.

pC [in]

Type: **const** D3DXQUATERNION*

Pointer to a source D3DXQUATERNION structure.

t [in]

Type: FLOAT

Parameter that indicates how far to interpolate between the quaternions.

Return value

Type: D3DXQUATERNION*

Pointer to a D3DXQUATERNION structure that is the result of the spherical quadrangle interpolation.

Remarks

This function uses the following sequence of spherical linear interpolation operations:

```
Slerp(Slerp(pQ1, pC, t), Slerp(pA, pB, t), 2t(1 - t))
```

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXQuaternionSquad function can be used as a parameter for another function.

Use D3DXQuaternionNormalize for any quaternion input that is not already normalized.

Requirements

Requirement	Value

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXQuaternionSquadSetup function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Sets up control points for spherical quadrangle interpolation.

Syntax

```C++
void D3DXQuaternionSquadSetup(
  _In_       D3DXQUATERNION *pAOut,
  _In_       D3DXQUATERNION *pBOut,
  _In_       D3DXQUATERNION *pCOut,
  _In_ const D3DXQUATERNION *pQ0,
  _In_ const D3DXQUATERNION *pQ1,
  _In_ const D3DXQUATERNION *pQ2,
  _In_ const D3DXQUATERNION *pQ3
);
```

Parameters

pAOut [in]

Type: **D3DXQUATERNION***

Pointer to AOut.

pBOut [in]

Type: **D3DXQUATERNION***

Pointer to BOut.

pCOut [in]

Type: **D3DXQUATERNION***

Pointer to COut.

pQ0 [in]

Type: **const** D3DXQUATERNION*

Pointer to the input control point, Q0.

pQ1 [in]

Type: **const** D3DXQUATERNION*

Pointer to the input control point, Q1.

pQ2 [in]

Type: **const** D3DXQUATERNION*

Pointer to the input control point, Q2.

pQ3 [in]

Type: **const** D3DXQUATERNION*

Pointer to the input control point, Q3.

Return value

None.

Remarks

This function takes four control points, which are supplied to the inputs pQ0, pQ1, pQ2, and pQ3. The function then alters these values to find a curve that flows along the shortest path. The values of q0, q2, and q3 are calculated as shown below.

```
q0 = |Q0 + Q1| < |Q0 - Q1| ? -Q0 : Q0
q2 = |Q1 + Q2| < |Q1 - Q2| ? -Q2 : Q2
q3 = |Q2 + Q3| < |Q2 - Q3| ? -Q3 : Q3
```

Having calculated the new Q values, the values for AOut, BOut, and COut are calculated as follows:

AOut = q1 * $e^{[-0.25\ *(\ Ln[Exp(q1)*q2]\ +\ Ln[Exp(q1)*q0]\ )\ ]}$

BOut = q2 * $e^{[-0.25\ *(\ Ln[Exp(q2)*q3]\ +\ Ln[Exp(q2)*q1]\ )\ ]}$

COut = q2

> ⓘ **Note**
>
> Ln is the API method **D3DXQuaternionLn** and Exp is the API method **D3DXQuaternionExp**.

Use **D3DXQuaternionNormalize** for any quaternion input that is not already normalized.

Examples

The following example shows how to use a set of quaternion keys (Q0, Q1, Q2, Q3) to compute the inner quadrangle points (A, B, C). This ensures that the tangents are continuous across adjacent segments.

```
      A     B
Q0    Q1    Q2    Q3
```

The following code example demonstrates how you can interpolate between Q1 and Q2.

```
// Rotation about the z-axis
D3DXQUATERNION Q0 = D3DXQUATERNION(0,  0, 0.707f, -.707f);
D3DXQUATERNION Q1 = D3DXQUATERNION(0,  0, 0.000f, 1.000f);
D3DXQUATERNION Q2 = D3DXQUATERNION(0,  0, 0.707f, 0.707f);
D3DXQUATERNION Q3 = D3DXQUATERNION(0,  0, 1.000f, 0.000f);
D3DXQUATERNION A, B, C, Qt;
FLOAT time = 0.5f;

D3DXQuaternionSquadSetup(&A, &B, &C, &Q0, &Q1, &Q2, &Q3);
D3DXQuaternionSquad(&Qt, &Q1, &A, &B, &C, time);
```

> ⓘ **Note**
>
> - C is +/- Q2 depending on the result of the function.
> - Qt is the result of the function.

The result is a rotation of 45 degrees around the z-axis for time = 0.5.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXQuaternionToAxisAngle function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Computes a quaternion's axis and angle of rotation.

Syntax

C++

```
void D3DXQuaternionToAxisAngle(
  _In_    const D3DXQUATERNION *pQ,
  _Inout_       D3DXVECTOR3    *pAxis,
  _Inout_       FLOAT          *pAngle
);
```

Parameters

pQ [in]

Type: **const** D3DXQUATERNION*

Pointer to the source D3DXQUATERNION.

pAxis [in, out]

Type: D3DXVECTOR3*

This function returns a pointer to a D3DXVECTOR3 that identifies the quaternion's axis of rotation.

pAngle [in, out]

Type: FLOAT*

This function returns a pointer to a FLOAT value that identifies the quaternion's angle of rotation in radians.

Return value

No return value.

Remarks

Use **D3DXQuaternionNormalize** for any quaternion input that is not already normalized.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXSHAdd function (D3DX10.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated and is not supported for Windows Store apps.

> **Note**
>
> Instead of using this function, we recommend that you use the **Spherical Harmonics Math** library function **XMSHAdd**.

Adds two spherical harmonic (SH) vectors; in other words, pOut[i] = pA[i] + pB[i].

Syntax

```C++
FLOAT* D3DXSHAdd(
  _In_       FLOAT *pOut,
  _In_       UINT  Order,
  _In_ const FLOAT *pA,
  _In_ const FLOAT *pB
);
```

Parameters

pOut [in]

Type: **FLOAT***

Pointer to SH output coefficients. The evaluation generates $Order^2$ coefficients. See Remarks.

Order [in]

Type: **UINT**

Order of the SH evaluation. Must be in the range of D3DXSH_MINORDER to D3DXSH_MAXORDER, inclusive. The evaluation generates $Order^2$ coefficients. The

degree of the evaluation is Order - 1.

pA [in]

Type: **const** **FLOAT***

Pointer to the first SH vector.

pB [in]

Type: **const** **FLOAT***

Pointer to the second SH vector.

Return value

Type: **FLOAT***

Pointer to SH output coefficients.

Remarks

Each coefficient of the basis function Y_{lm} is stored at memory location $l^2 + m + l$, where:

- l is the degree of the basis function.
- m is the basis function index for the given l value and ranges from -l to l, inclusive.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

Math Functions

D3DXSHDot function (D3DX10.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated and is not supported for Windows Store apps.

> **Note**
>
> Instead of using this function, we recommend that you use the **Spherical Harmonics Math** library function **XMSHDot**.

Computes the dot product of two spherical harmonic (SH) vectors.

Syntax

```C++
FLOAT D3DXSHDot(
  _In_       UINT  Order,
  _In_ const FLOAT *pA,
  _In_ const FLOAT *pB
);
```

Parameters

Order [in]

Type: **UINT**

Order of the spherical harmonic (SH) evaluation. Must be in the range of D3DXSH_MINORDER to D3DXSH_MAXORDER, inclusive. The evaluation generates $Order^2$ coefficients. The degree of the evaluation is Order - 1.

pA [in]

Type: **const FLOAT***

Pointer to the first SH vector.

pB [in]

Type: **const** **FLOAT***

Pointer to the second SH vector.

Return value

Type: **FLOAT**

SH output coefficients.

Remarks

Each coefficient of the basis function Y_{lm} is stored at memory location $l^2 + m + l$, where:

- l is the degree of the basis function.
- m is the basis function index for the given l value and ranges from -l to l, inclusive.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

Math Functions

D3DXSHEvalConeLight function (D3DX10.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated and is not supported for Windows Store apps.

> **Note**
>
> Instead of using this function, we recommend that you use the **Spherical Harmonics Math** library function **XMSHEvalConeLight**.

Evaluates a light that is a cone of constant intensity and returns spectral spherical harmonic (SH) data.

Syntax

C++

```
HRESULT D3DXSHEvalConeLight(
  _In_       UINT        Order,
  _In_ const D3DXVECTOR3 *pDir,
  _In_       FLOAT       Radius,
  _In_       FLOAT       RIntensity,
  _In_       FLOAT       GIntensity,
  _In_       FLOAT       BIntensity,
  _In_       FLOAT       *pROut,
  _In_       FLOAT       *pGOut,
  _In_       FLOAT       *pBOut
);
```

Parameters

Order [in]

Type: **UINT**

Order of the SH evaluation. Must be in the range of D3DXSH_MINORDER to D3DXSH_MAXORDER, inclusive. The evaluation generates $Order^2$ coefficients. The degree of the evaluation is Order - 1.

pDir [in]

Type: **const** **D3DXVECTOR3***

Pointer to the (x, y, z) hemisphere axis direction vector in which to evaluate the SH basis functions. See Remarks.

Radius [in]

Type: **FLOAT**

Radius of cone in radians.

RIntensity [in]

Type: **FLOAT**

The red intensity of the light.

GIntensity [in]

Type: **FLOAT**

The green intensity of the light.

BIntensity [in]

Type: **FLOAT**

The blue intensity of the light.

pROut [in]

Type: **FLOAT***

Pointer to the output SH vector for the red component.

pGOut [in]

Type: **FLOAT***

Pointer to the output SH vector for the green component.

pBOut [in]

Type: **FLOAT***

Pointer to the output SH vector for the blue component.

Return value

Type: **HRESULT**

If the function succeeds, the return value is D3D_OK. If the function fails, the return value can be: D3DERR_INVALIDCALL.

Remarks

Evaluates a light that is a cone of constant intensity and returns spectral SH data. The output vector is computed so that if the intensity ratio R/G/B is equal to 1, the exit radiance of a point directly under the light (oriented in the cone direction on a diffuse object with an albedo of 1) would be 1.0. This will compute three spectral samples; pROut will be returned, while pGOut and pBOut may be returned.

On the sphere with unit radius, as shown in the following illustration, direction can be specified simply with theta, the angle about the z-axis in the right-handed direction, and phi, the angle from z.

The following equations show the relationship between Cartesian (x, y, z) and spherical (theta, phi) coordinates on the unit sphere. The angle theta varies over the range of 0 to 2 pi, while phi varies from 0 to pi.

$$x = \cos\theta \sin\phi$$
$$y = \sin\theta \sin\phi$$
$$z = \cos\phi$$

$$\theta = \tan^{-1}(y/x)$$
$$\phi = \cos^{-1}(z)$$

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

Math Functions

D3DXSHEvalDirection function (D3DX10.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated and is not supported for Windows Store apps.

> **Note**
>
> Instead of using this function, we recommend that you use the **Spherical Harmonics Math** library function **XMSHEvalDirection**.

Evaluates the spherical harmonic (SH) basis functions from an input direction vector.

Syntax

C++

```
FLOAT* D3DXSHEvalDirection(
  _In_       FLOAT       *pOut,
  _In_       UINT        Order,
  _In_ const D3DXVECTOR3 *pDir
);
```

Parameters

pOut [in]

Type: **FLOAT***

Pointer to Spherical harmonic (SH) output coefficients. The evaluation generates $Order^2$ coefficients. See Remarks.

Order [in]

Type: **UINT**

Order of the SH evaluation. Must be in the range of D3DXSH_MINORDER to D3DXSH_MAXORDER, inclusive. The evaluation generates Order^2 coefficients. The degree of the evaluation is Order - 1.

pDir [in]

Type: **const** D3DXVECTOR3*

(x, y, z) direction vector in which to evaluate the SH basis functions. Must be normalized. See Remarks.

Return value

Type: FLOAT*

Pointer to SH output coefficients. See Remarks.

Remarks

Each coefficient of the basis function Y_{lm} is stored at memory location $l^2 + m + l$, where:

- l is the degree of the basis function.
- m is the basis function index for the given l value and ranges from -l to l, inclusive.

On the sphere with unit radius, as shown in the following illustration, direction can be specified simply with theta, the angle about the z-axis in the right-handed direction, and phi, the angle from z.

The following equations show the relationship between Cartesian (x, y, z) and spherical (theta, phi) coordinates on the unit sphere. The angle theta varies over the range of 0 to 2 pi, while phi varies from 0 to pi.

$$x = \cos\theta \sin\phi$$
$$y = \sin\theta \sin\phi$$
$$z = \cos\phi$$

$$\theta = \tan^{-1}(y/x)$$
$$\phi = \cos^{-1}(z)$$

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

Math Functions

D3DXSHEvalDirectionalLight function (D3DX10.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated and is not supported for Windows Store apps.

> **Note**
>
> Instead of using this function, we recommend that you use the **Spherical Harmonics Math** library function **XMSHEvalDirectionalLight**.

Evaluates a directional light and returns spectral spherical harmonic (SH) data.

Syntax

C++

```
HRESULT D3DXSHEvalDirectionalLight(
  _In_       UINT        Order,
  _In_ const D3DXVECTOR3 *pDir,
  _In_       FLOAT       RIntensity,
  _In_       FLOAT       GIntensity,
  _In_       FLOAT       BIntensity,
  _In_       FLOAT       *pROut,
  _In_       FLOAT       *pGOut,
  _In_       FLOAT       *pBOut
);
```

Parameters

Order [in]

Type: **UINT**

Order of the SH evaluation. Must be in the range of D3DXSH_MINORDER to D3DXSH_MAXORDER, inclusive. The evaluation generates $Order^2$ coefficients. The degree of the evaluation is Order - 1.

pDir [in]

Type: **const** D3DXVECTOR3*

Pointer to the (x, y, z) hemisphere axis direction vector in which to evaluate the SH basis functions. See Remarks.

RIntensity [in]

Type: FLOAT

The red intensity of the light.

GIntensity [in]

Type: FLOAT

The green intensity of the light.

BIntensity [in]

Type: FLOAT

The blue intensity of the light.

pROut [in]

Type: FLOAT*

Pointer to the output SH vector for the red component.

pGOut [in]

Type: FLOAT*

Optional pointer to the output SH vector for the green component.

pBOut [in]

Type: FLOAT*

Optional pointer to the output SH vector for the blue component.

Return value

Type: HRESULT

If the function succeeds, the return value is D3D_OK. If the function fails, the return value can be: D3DERR_INVALIDCALL.

Remarks

The output vector is computed so that if the intensity ratio R/G/B is equal to 1, the resulting exit radiance of a point directly under the light on a diffuse object with an albedo of 1 would be 1.0. This will compute three spectral samples; pROut will be returned, while pGOut and pBOut may be returned.

On the sphere with unit radius, as shown in the following illustration, direction can be specified simply with theta, the angle about the z-axis in the right-handed direction, and phi, the angle from z.

The following equations show the relationship between Cartesian (x, y, z) and spherical (theta, phi) coordinates on the unit sphere. The angle theta varies over the range of 0 to 2 pi, while phi varies from 0 to pi.

$$x = \cos\theta \sin\phi$$
$$y = \sin\theta \sin\phi$$
$$z = \cos\phi$$

$$\theta = \tan^{-1}(y/x)$$
$$\phi = \cos^{-1}(z)$$

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

Math Functions

D3DXSHEvalHemisphereLight function (D3DX10.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated and is not supported for Windows Store apps.

> **Note**
>
> Instead of using this function, we recommend that you use the Spherical Harmonics Math library function **XMSHEvalHemisphereLight**.

Evaluates a light that is a linear interpolation between two colors over the sphere.

Syntax

C++

```
HRESULT D3DXSHEvalHemisphereLight(
  _In_       UINT        Order,
  _In_ const D3DXVECTOR3 *pDir,
  _In_       D3DXCOLOR   Top,
  _In_       D3DXCOLOR   Bottom,
  _In_       FLOAT       *pROut,
  _In_       FLOAT       *pGOut,
  _In_       FLOAT       *pBOut
);
```

Parameters

Order [in]

Type: UINT

Order of the spherical harmonic (SH) evaluation. Must be in the range of D3DXSH_MINORDER to D3DXSH_MAXORDER, inclusive. The evaluation generates $Order^2$ coefficients. The degree of the evaluation is Order - 1.

pDir [in]

Type: **const** D3DXVECTOR3*

Pointer to the (x, y, z) hemisphere axis direction vector in which to evaluate the SH basis functions. See Remarks.

Top [in]

Type: D3DXCOLOR

The sky color.

Bottom [in]

Type: D3DXCOLOR

The ground color.

pROut [in]

Type: FLOAT*

Pointer to the output SH vector for the red component.

pGOut [in]

Type: FLOAT*

Pointer to the output SH vector for the green component.

pBOut [in]

Type: FLOAT*

Pointer to the output SH vector for the blue component.

Return value

Type: HRESULT

If the function succeeds, the return value is D3D_OK. If the function fails, the return value can be: D3DERR_INVALIDCALL.

Remarks

The interpolation is done linearly between the two points, not over the surface of the sphere (that is, if the axis was (0,0,1) it is linear in Z, not in the azimuthal angle). The resulting spherical lighting function is normalized so that a point on a perfectly diffuse surface with no shadowing and a normal pointed in the direction pDir would result in exit radiance with a value of 1 (if the top color was white and the bottom color was black). This is a very simple model where Top represents the intensity of the "sky" and Bottom represents the intensity of the "ground."

On the sphere with unit radius, as shown in the following illustration, direction can be specified simply with theta, the angle about the z-axis in the right-handed direction, and phi, the angle from z.

The following equations show the relationship between Cartesian (x, y, z) and spherical (theta, phi) coordinates on the unit sphere. The angle theta varies over the range of 0 to 2 pi, while phi varies from 0 to pi.

$$x = \cos\theta \sin\phi$$
$$y = \sin\theta \sin\phi$$
$$z = \cos\phi$$

$$\theta = \tan^{-1}(y/x)$$
$$\phi = \cos^{-1}(z)$$

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

Math Functions

D3DXSHEvalSphericalLight function (D3DX10.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated and is not supported for Windows Store apps.

> **Note**
>
> Instead of using this function, we recommend that you use the **Spherical Harmonics Math** library function **XMSHEvalSphericalLight**.

Evaluates a spherical light and returns spectral spherical harmonic (SH) data.

Syntax

C++

```
HRESULT D3DXSHEvalSphericalLight(
  _In_       UINT        Order,
  _In_ const D3DXVECTOR3 *pPos,
  _In_       FLOAT       Radius,
  _In_       FLOAT       RIntensity,
  _In_       FLOAT       GIntensity,
  _In_       FLOAT       BIntensity,
  _In_       FLOAT       *pROut,
  _In_       FLOAT       *pGOut,
  _In_       FLOAT       *pBOut
);
```

Parameters

Order [in]

Type: **UINT**

Order of the SH evaluation. Must be in the range of D3DXSH_MINORDER to D3DXSH_MAXORDER, inclusive. The evaluation generates $Order^2$ coefficients. The

degree of the evaluation is Order - 1.

pPos [in]

Type: **const** **D3DXVECTOR3***

Pointer to the light position.

Radius [in]

Type: **FLOAT**

Radius of spherical light source.

RIntensity [in]

Type: **FLOAT**

The red intensity of the light.

GIntensity [in]

Type: **FLOAT**

The green intensity of the light.

BIntensity [in]

Type: **FLOAT**

The blue intensity of the light.

pROut [in]

Type: **FLOAT***

Pointer to the output SH vector for the red component.

pGOut [in]

Type: **FLOAT***

Pointer to the output SH vector for the green component.

pBOut [in]

Type: **FLOAT***

Pointer to the output SH vector for the blue component.

Return value

Type: HRESULT

If the function succeeds, the return value is D3D_OK. If the function fails, the return value can be: D3DERR_INVALIDCALL.

Remarks

Evaluates a spherical light and returns spectral SH data. There is no normalization of the intensity of the light like there is for directional lights, so care has to be taken when specifying the intensities. This will compute three spectral samples; pROut will be returned, while pGOut and pBOut may be returned.

On the sphere with unit radius, as shown in the following illustration, direction can be specified simply with theta, the angle about the z-axis in the right-handed direction, and phi, the angle from z.

The following equations show the relationship between Cartesian (x, y, z) and spherical (theta, phi) coordinates on the unit sphere. The angle theta varies over the range of 0 to 2 pi, while phi varies from 0 to pi.

$$x = \cos\theta \sin\phi$$
$$y = \sin\theta \sin\phi$$
$$z = \cos\phi$$

$$\theta = \tan^{-1}(y/x)$$
$$\phi = \cos^{-1}(z)$$

Requirements

Requirement	Value

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

Math Functions

D3DXSHMultiply2 function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated and is not supported for Windows Store apps.

> **Note**
>
> Instead of using this function, we recommend that you use the **Spherical Harmonics Math** library function **XMSHMultiply2**.

Computes the product of two spherical harmonics functions (*f* and *g*). Both functions are of order N = 2.

Syntax

```
C++

FLOAT* D3DXSHMultiply2(
  _In_       FLOAT *pOut,
  _In_ const FLOAT *pF,
  _In_ const FLOAT *pG
);
```

Parameters

pOut [in]

Type: **FLOAT***

Pointer to the output SH coefficients — the basis function Y_{lm} is stored at $l^2 + m + l$. The order *N* determines the length of the array, where there should always be N^2 coefficients.

pF [in]

Type: **const** FLOAT*

Input SH coefficients for first function.

pG [in]

Type: **const** FLOAT*

Second set of input SH coefficients.

Return value

Type: FLOAT*

Pointer to SH output coefficients.

Remarks

The product of two SH functions of order N = 2 generates an SH function of order 2 × N - 1 = 3, but the results are truncated. This means that the product commutes ($f \times g = g \times f$) but doesn't associate ($f \times (g \times h) \neq (f \times g) \times h$).

This function uses the following equation:

```
pOut[i] = int(y_i(s) * f(s) * g(s))
```

where y_i(s) is the ith SH basis function, and where f(s) and g(s) use the following SH function:

```
sum_i(y_i(s)*c_i)
```

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXSHMultiply3 function

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated and is not supported for Windows Store apps.

> **Note**
>
> Instead of using this function, we recommend that you use the **Spherical Harmonics Math** library function **XMSHMultiply3**.

Computes the product of two spherical harmonics functions (*f* and *g*). Both functions are of order N = 3.

Syntax

C++

```
FLOAT* D3DXSHMultiply3(
  _In_       FLOAT *pOut,
  _In_ const FLOAT *pF,
  _In_ const FLOAT *pG
);
```

Parameters

pOut [in]

Type: **FLOAT***

Pointer to the output SH coefficients — the basis function Y_{lm} is stored at $l^2 + m + l$. The order *N* determines the length of the array, where there should always be N^2 coefficients.

pF [in]

Type: **const FLOAT***

Input SH coefficients for first function.

pG [in]

Type: **const** FLOAT*

Second set of input SH coefficients.

Return value

Type: FLOAT*

Pointer to SH output coefficients.

Remarks

The product of two SH functions of order N = 3 generates an SH function of order 2 × N - 1 = 5, but the results are truncated. This means that the product commutes ($f \times g = g \times f$) but doesn't associate ($f \times (g \times h) \neq (f \times g) \times h$).

This function uses the following equation:

```
pOut[i] = int(y_i(s) * f(s) * g(s))
```

where y_i(s) is the ith SH basis function, and where f(s) and g(s) use the following SH function:

```
sum_i(y_i(s)*c_i)
```

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXSHMultiply4 function

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated and is not supported for Windows Store apps.

> **Note**
>
> Instead of using this function, we recommend that you use the **Spherical Harmonics Math** library function **XMSHMultiply4**.

Computes the product of two spherical harmonics functions (*f* and *g*). Both functions are of order N = 4.

Syntax

C++

```
FLOAT* D3DXSHMultiply4(
  _In_       FLOAT *pOut,
  _In_ const FLOAT *pF,
  _In_ const FLOAT *pG
);
```

Parameters

pOut [in]

Type: **FLOAT***

Pointer to the output SH coefficients — basis function Y_{lm} is stored at $l^2 + m + l$. The order *N* determines the length of the array, where there should always be N^2 coefficients.

pF [in]

Type: **const FLOAT***

Input SH coefficients for first function.

pG [in]

Type: **const** **FLOAT***

Second set of input SH coefficients.

Return value

Type: **FLOAT***

Pointer to SH output coefficients.

Remarks

The product of two SH functions of order N = 4 generates an SH function of order 2 × N - 1 = 7, but the results are truncated. This means that the product commutes ($f \times g = g \times f$) but doesn't associate ($f \times (g \times h) \neq (f \times g) \times h$).

This function uses the following equation:

```
pOut[i] = int(y_i(s) * f(s) * g(s))
```

where y_i(s) is the ith SH basis function, and where f(s) and g(s) use the following SH function:

```
sum_i(y_i(s)*c_i)
```

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXSHMultiply5 function

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated and is not supported for Windows Store apps.

> **Note**
>
> Instead of using this function, we recommend that you use the **Spherical Harmonics Math** library function **XMSHMultiply5**.

Computes the product of two spherical harmonics functions (f and g). Both functions are of order N = 5.

Syntax

C++

```
FLOAT* D3DXSHMultiply5(
  _In_       FLOAT *pOut,
  _In_ const FLOAT *pF,
  _In_ const FLOAT *pG
);
```

Parameters

pOut [in]

Type: **FLOAT***

Pointer to the output SH coefficients — the basis function Y_{lm} is stored at $l^2 + m + l$. The order N determines the length of the array, where there should always be N^2 coefficients.

pF [in]

Type: **const FLOAT***

Input SH coefficients for first function.

pG [in]

Type: **const** **FLOAT***

Second set of input SH coefficients.

Return value

Type: **FLOAT***

Pointer to SH output coefficients.

Remarks

The product of two SH functions of order N = 5 generates an SH function of order 2 × N - 1 = 9, but the results are truncated. This means that the product commutes ($f \times g = g \times f$) but doesn't associate ($f \times (g \times h) \neq (f \times g) \times h$).

This function uses the following equation:

```
pOut[i] = int(y_i(s) * f(s) * g(s))
```

where y_i(s) is the ith SH basis function, and where f(s) and g(s) use the following SH function:

```
sum_i(y_i(s)*c_i)
```

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXSHMultiply6 function

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated and is not supported for Windows Store apps.

> **Note**
>
> Instead of using this function, we recommend that you use the **Spherical Harmonics Math** library function **XMSHMultiply6**.

Computes the product of two spherical harmonics functions (f and g). Both functions are of order N = 6.

Syntax

C++

```
FLOAT* D3DXSHMultiply6(
  _In_       FLOAT *pOut,
  _In_ const FLOAT *pF,
  _In_ const FLOAT *pG
);
```

Parameters

pOut [in]

Type: **FLOAT***

Pointer to the output SH coefficients — the basis function Y_{lm} is stored at $l^2 + m + l$. The order N determines the length of the array, where there should always be N^2 coefficients.

pF [in]

Type: **const FLOAT***

Input SH coefficients for first function.

pG [in]

Type: **const** **FLOAT***

Second set of input SH coefficients.

Return value

Type: **FLOAT***

Pointer to SH output coefficients.

Remarks

The product of two SH functions of order N = 6 generates an SH function of order 2 × N - 1 = 11, but the results are truncated. This means that the product commutes ($f \times g = g \times f$) but doesn't associate ($f \times (g \times h) \neq (f \times g) \times h$).

This function uses the following equation:

```
pOut[i] = int(y_i(s) * f(s) * g(s))
```

where y_i(s) is the ith SH basis function, and where f(s) and g(s) use the following SH function:

```
sum_i(y_i(s)*c_i)
```

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DX10SHProjectCubeMap function

Article • 03/15/2021 • 2 minutes to read

Projects a function represented in a cube map into spherical harmonics.

Syntax

```C++
HRESULT D3DX10SHProjectCubeMap(
   UINT            Order,
   ID3D10Texture2D *pCubeMap,
   FLOAT           *pROut,
   FLOAT           *pGOut,
   FLOAT           *pBOut
);
```

Parameters

Order

Type: **UINT**

Order of the SH evaluation, generates Order^2 coefs, degree is Order-1.

pCubeMap

Type: **ID3D10Texture2D***

Cubemap that is going to be projected into spherical harmonics. See **ID3D10Texture2D**.

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 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10Tex.h
Library	D3DX10.lib

See also

Math Functions

D3DXSHRotate function (D3DX10.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated and is not supported for Windows Store apps.

> **Note**
>
> Instead of using this function, we recommend that you use the **Spherical Harmonics Math** library function **XMSHRotate**.

Rotates the spherical harmonic (SH) vector by the given matrix.

Syntax

C++

```
FLOAT* D3DXSHRotate(
  _In_       FLOAT      *pOut,
  _In_       UINT       Order,
  _In_ const D3DXMATRIX *pMatrix,
  _In_ const FLOAT      *pIn
);
```

Parameters

pOut [in]

Type: **FLOAT***

Pointer to Spherical harmonic (SH) output coefficients. The evaluation generates $Order^2$ coefficients. This pointer should not alias with pIn. See Remarks.

Order [in]

Type: **UINT**

Order of the SH evaluation. Must be in the range of D3DXSH_MINORDER to D3DXSH_MAXORDER, inclusive. The evaluation generates $Order^2$ coefficients. The

degree of the evaluation is Order - 1.

pMatrix [in]

Type: **const** **D3DXMATRIX***

Pointer to the rotation matrix. The rotation sub-matrix must be orthogonal, with a unit determinant.

pIn [in]

Type: **const** **FLOAT***

Pointer to rotated SH coefficients.

Return value

Type: **FLOAT***

Pointer to SH output coefficients.

Remarks

Each coefficient of the basis function Y_{lm} is stored at memory location $l^2 + m + l$, where:

- l is the degree of the basis function.
- m is the basis function index for the given l value and ranges from -l to l, inclusive.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

Math Functions

D3DXSHRotateZ function (D3DX10.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated and is not supported for Windows Store apps.

> **Note**
>
> Instead of using this function, we recommend that you use the **Spherical Harmonics Math** library function **XMSHRotateZ**.

Rotates the spherical harmonic (SH) vector in the z-axis by the given angle.

Syntax

```
C++

FLOAT* D3DXSHRotateZ(
  _In_       FLOAT *pOut,
  _In_       UINT  Order,
  _In_       FLOAT Angle,
  _In_ const FLOAT *pIn
);
```

Parameters

pOut [in]

Type: **FLOAT***

Pointer to Spherical harmonic (SH) output coefficients. The evaluation generates $Order^2$ coefficients. This pointer should not alias with pIn. See Remarks.

Order [in]

Type: **UINT**

Order of the SH evaluation. Must be in the range of D3DXSH_MINORDER to D3DXSH_MAXORDER, inclusive. The evaluation generates $Order^2$ coefficients. The

degree of the evaluation is Order - 1.

Angle [in]

Type: **FLOAT**

Rotation angle in radians. The rotation is performed around the z-axis.

pIn [in]

Type: **const FLOAT***

Pointer to rotated SH coefficients.

Return value

Type: **FLOAT***

Pointer to SH output coefficients.

Remarks

Each coefficient of the basis function Y_{lm} is stored at memory location $l^2 + m + l$, where:

- l is the degree of the basis function.
- m is the basis function index for the given l value and ranges from -l to l, inclusive.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

Math Functions

D3DXSHScale function (D3DX10.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX (D3DX 9, D3DX 10, and D3DX 11) utility library is deprecated and is not supported for Windows Store apps.

> **Note**
>
> Instead of using this function, we recommend that you use the **Spherical Harmonics Math** library function **XMSHScale**.

Scales a spherical harmonic (SH) vector; in other words, pOut[i] = pA[i]*Scale.

Syntax

C++

```
FLOAT* D3DXSHScale(
  _In_       FLOAT *pOut,
  _In_       UINT  Order,
  _In_ const FLOAT *pIn,
  _In_ const FLOAT Scale
);
```

Parameters

pOut [in]

Type: **FLOAT***

Pointer to Spherical harmonic (SH) output coefficients. The evaluation generates $Order^2$ coefficients. See Remarks.

Order [in]

Type: **UINT**

Order of the SH evaluation. Must be in the range of D3DXSH_MINORDER to D3DXSH_MAXORDER, inclusive. The evaluation generates $Order^2$ coefficients. The

degree of the evaluation is Order - 1.

pIn [in]

Type: **const** **FLOAT***

Pointer to the SH vector to scale.

Scale [in]

Type: **const** **FLOAT**

Pointer to the scale value.

Return value

Type: **FLOAT***

Pointer to SH output coefficients.

Remarks

Each coefficient of the basis function Y_{lm} is stored at memory location $l^2 + m + l$, where:

- l is the degree of the basis function.
- m is the basis function index for the given l value and ranges from -l to l, inclusive.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

Math Functions

D3DXVec2BaryCentric function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Returns a point in Barycentric coordinates, using the specified 2D vectors.

Syntax

```C++
D3DXVECTOR2* D3DXVec2BaryCentric(
  _In_       D3DXVECTOR2 *pOut,
  _In_ const D3DXVECTOR2 *pV1,
  _In_ const D3DXVECTOR2 *pV2,
  _In_ const D3DXVECTOR2 *pV3,
  _In_       FLOAT       f,
  _In_       FLOAT       g
);
```

Parameters

pOut [in]

Type: **D3DXVECTOR2***

Pointer to the **D3DXVECTOR2** that is the result of the operation.

pV1 [in]

Type: **const D3DXVECTOR2***

Pointer to a source D3DXVECTOR2 structure.

pV2 [in]

Type: **const D3DXVECTOR2***

Pointer to a source D3DXVECTOR2 structure.

pV3 [in]

Type: **const** **D3DXVECTOR2***

Pointer to a source D3DXVECTOR2 structure.

f [in]

Type: **FLOAT**

Weighting factor. See Remarks.

g [in]

Type: **FLOAT**

Weighting factor. See Remarks.

Return value

Type: **D3DXVECTOR2***

Pointer to a D3DXVECTOR2 structure in Barycentric coordinates.

Remarks

The D3DXVec2BaryCentric function provides a way to understand points in and around a triangle, independent of where the triangle is actually located. This function returns the resulting point by using the following equation: V1 + f(V2-V1) + g(V3-V1).

Any point in the plane V1V2V3 can be represented by the Barycentric coordinate (f,g).The parameter f controls how much V2 gets weighted into the result, and the parameter g controls how much V3 gets weighted into the result. Lastly, 1-f-g controls how much V1 gets weighted into the result.

Note the following relations.

- If (f>=0 &, & g>=0 &, & 1-f-g>=0), the point is inside the triangle V1V2V3.
- If (f==0 &, & g>=0 &, & 1-f-g>=0), the point is on the line V1V3.
- If (f>=0 &, & g==0 &, & 1-f-g>=0), the point is on the line V1V2.
- If (f>=0 &, & g>=0 &, & 1-f-g==0), the point is on the line V2V3.

Barycentric coordinates are a form of general coordinates. In this context, using Barycentric coordinates represents a change in coordinate systems. What holds true for Cartesian coordinates holds true for Barycentric coordinates.

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXVec2BaryCentric function can be used as a parameter for another function.

Barycentric coordinates define a point inside a triangle in terms of the triangle's vertices. For a more in-depth description of barycentric coordinates, see Mathworld's Barycentric Coordinates Description.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXVec2CatmullRom function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Performs a Catmull-Rom interpolation, using the specified 2D vectors.

Syntax

C++

```
D3DXVECTOR2* D3DXVec2CatmullRom(
  _Inout_       D3DXVECTOR2 *pOut,
  _In_    const D3DXVECTOR2 *pV0,
  _In_    const D3DXVECTOR2 *pV1,
  _In_    const D3DXVECTOR2 *pV2,
  _In_    const D3DXVECTOR2 *pV3,
  _In_          FLOAT       s
);
```

Parameters

pOut [in, out]

Type: **D3DXVECTOR2***

Pointer to the **D3DXVECTOR2** that is the result of the operation.

pV0 [in]

Type: **const D3DXVECTOR2***

Pointer to a source D3DXVECTOR2 structure, a position vector.

pV1 [in]

Type: **const D3DXVECTOR2***

Pointer to a source D3DXVECTOR2 structure, a position vector.

pV2 [in]

Type: **const** D3DXVECTOR2*

Pointer to a source D3DXVECTOR2 structure, a position vector.

pV3 [in]

Type: **const** D3DXVECTOR2*

Pointer to a source D3DXVECTOR2 structure, a position vector.

s [in]

Type: FLOAT

Weighting factor. See Remarks.

Return value

Type: D3DXVECTOR2*

Pointer to a D3DXVECTOR2 structure that is the result of the Catmull-Rom interpolation.

Remarks

Given four points (p1, p2, p3, p4), find a function Q(s) such that:

```
Q(s) is a cubic function.
Q(s) interpolates between p2 and p3 as s ranges from 0 to 1.
Q(s) is parallel to the line joining p1 to p3 when s is 0.
Q(s) is parallel to the line joining p2 to p4 when s is 1.
```

The Catmull-Rom spline can be derived from the Hermite spline by setting:

```
v1 = p2
v2 = p3
t1 = (p3 - p1) / 2
t2 = (p4 - p2) / 2
```

where:

v1 is the contents of pV0.

v2 is the contents of pV1.

p3 is the contents of pV2.

p4 is the contents of pV3.

Using the Hermite spline equation:

```
Q(s) = (2s3 - 3s2 + 1)v1 + (-2s3 + 3s2)v2 + (s3 - 2s2 + s)t1 + (s3 - s2)t2
```

and substituting for v1, v2, t1, t2 yields:

```
Q(s) = (2s3 - 3s2 + 1)p2 + (-2s3 + 3s2)p3 + (s3 - 2s2 + s)(p3 - p1) / 2 +
(s3 - s2)(p4 - p2)/2
```

This can be rearranged as:

```
Q(s) = [(-s3 + 2s2 - s)p1 + (3s3 - 5s2 + 2)p2 + (-3s3 + 4s2 + s)p3 + (s3 -
s2)p4] / 2
```

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXVec2Hermite function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Performs a Hermite spline interpolation, using the specified 2D vectors.

Syntax

```C++
D3DXVECTOR2* D3DXVec2Hermite(
  _Inout_       D3DXVECTOR2 *pOut,
  _In_    const D3DXVECTOR2 *pV1,
  _In_    const D3DXVECTOR2 *pT1,
  _In_    const D3DXVECTOR2 *pV2,
  _In_    const D3DXVECTOR2 *pT2,
  _In_          FLOAT       s
);
```

Parameters

pOut [in, out]

Type: **D3DXVECTOR2***

Pointer to the **D3DXVECTOR2** that is the result of the operation.

pV1 [in]

Type: **const D3DXVECTOR2***

Pointer to a source D3DXVECTOR2 structure, a position vector.

pT1 [in]

Type: **const D3DXVECTOR2***

Pointer to a source D3DXVECTOR2 structure, a tangent vector.

pV2 [in]

Type: **const** D3DXVECTOR2*

Pointer to a source D3DXVECTOR2 structure, a position vector.

pT2 [in]

Type: **const** D3DXVECTOR2*

Pointer to a source D3DXVECTOR2 structure, a tangent vector.

s [in]

Type: FLOAT

Weighting factor. See Remarks.

Return value

Type: D3DXVECTOR2*

Pointer to a D3DXVECTOR2 structure that is the result of the Hermite spline interpolation.

Remarks

The **D3DXVec2Hermite** function interpolates from (positionA, tangentA) to (positionB, tangentB) using Hermite spline interpolation.

The spline interpolation is a generalization of the ease-in, ease-out spline. The ramp is a function of Q(s) with the following properties.

$Q(s) = As^3 + Bs^2 + Cs + D$ (and therefore, $Q'(s) = 3As^2 + 2Bs + C$)

a) Q(0) = v1, so Q'(0) = t1

b) Q(1) = v2, so Q'(1) = t2

v1 is the contents of pV1, v2 in the contents of pV2, t1 is the contents of pT1, and t2 is the contents of pT2.

These properties are used to solve for A, B, C, D.

```
D = v1  (from a)
C = t1  (from a)
3A + 2B = t2 - t-1 (substituting for C)
A + B = v2 - v1 - t1 (substituting for C and D)
```

Plug in the solutions for A,B,C and D to generate Q(s).

```
A = 2v1 - 2v2 + t2 + t1
B = 3v2 - 3v1 - 2t1 - t2
C = t1
D = v1
```

This yields:

$Q(s) = (2v1 - 2v2 + t2 + t1)s^3 + (3v2 - 3v1 - 2t1 - t2)s^2 + t1s + v1$

Which can be rearranged as:

$Q(s) = (2s^3 - 3s^2 + 1)v1 + (-2s^3 + 3s^2)v2 + (s^3 - 2s^2 + s)t1 + (s^3 - s^2)t2$

Hermite splines are useful for controlling animation because the curve runs through all the control points. Also, because the position and tangent are explicitly specified at the ends of each segment, it is easy to create a C2 continuous curve as long as you make sure that your starting position and tangent match the ending values of the last segment.

The return value for this function is the same value returned in the pOut parameter. In this way, the **D3DXVec2Hermite** function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXVec2Normalize function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Returns the normalized version of a 2D vector.

Syntax

```C++
D3DXVECTOR2* D3DXVec2Normalize(
  _Inout_       D3DXVECTOR2 *pOut,
  _In_    const D3DXVECTOR2 *pV
);
```

Parameters

pOut [in, out]

Type: **D3DXVECTOR2***

Pointer to the **D3DXVECTOR2** that is the result of the operation.

pV [in]

Type: **const D3DXVECTOR2***

Pointer to the source D3DXVECTOR2 structure.

Return value

Type: **D3DXVECTOR2***

Pointer to a D3DXVECTOR2 structure that is the normalized version of the vector.

Remarks

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXVec2Normalize function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXVec2Transform function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Transforms a 2D vector by a given matrix.

Syntax

```C++
D3DXVECTOR4* D3DXVec2Transform(
  _Inout_       D3DXVECTOR4 *pOut,
  _In_    const D3DXVECTOR2 *pV,
  _In_    const D3DXMATRIX  *pM
);
```

Parameters

pOut [in, out]

Type: **D3DXVECTOR4***

Pointer to the **D3DXVECTOR4** structure that is the result of the operation.

pV [in]

Type: **const D3DXVECTOR2***

Pointer to the source **D3DXVECTOR2**.

pM [in]

Type: **const D3DXMATRIX***

Pointer to the source **D3DXMATRIX** structure.

Return value

Type: **D3DXVECTOR4***

Pointer to a D3DXVECTOR4 structure that is the transformed vector.

Remarks

This function transforms the vector pV (x, y, 0, 1) by the matrix pM.

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXVec2Transform function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXVec2TransformArray function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Transforms an array (x, y, 0, 1) by a given matrix.

Syntax

C++

```
D3DXVECTOR4* D3DXVec2TransformArray(
  _Inout_       D3DXVECTOR4 *pOut,
  _In_          UINT        OutStride,
  _In_    const D3DXVECTOR2 *pV,
  _In_          UINT        VStride,
  _In_    const D3DXMATRIX  *pM,
  _In_          UINT        n
);
```

Parameters

pOut [in, out]

Type: **D3DXVECTOR4***

Pointer to the **D3DXVECTOR4** structure that is the result of the operation.

OutStride [in]

Type: **UINT**

Stride between vectors in the output data stream.

pV [in]

Type: **const D3DXVECTOR2***

Pointer to the source D3DXVECTOR2.

VStride [in]

Type: UINT

Stride between vectors in the input data stream.

pM [in]

Type: **const** D3DXMATRIX*

Pointer to the source D3DXMATRIX structure.

n [in]

Type: UINT

Number of elements in the array.

Return value

Type: D3DXVECTOR4*

Pointer to a D3DXVECTOR4 structure that is the transformed array.

Remarks

This function transforms the array pV (x, y, 0, 1) by the matrix pM.

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXVec2Transform function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXVec2TransformCoord function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Transforms a 2D vector by a given matrix, projecting the result back into w = 1.

Syntax

```C++
D3DXVECTOR2* D3DXVec2TransformCoord(
  _Inout_       D3DXVECTOR2 *pOut,
  _In_    const D3DXVECTOR2 *pV,
  _In_    const D3DXMATRIX  *pM
);
```

Parameters

pOut [in, out]

Type: **D3DXVECTOR2***

Pointer to the **D3DXVECTOR2** that is the result of the operation.

pV [in]

Type: **const D3DXVECTOR2***

Pointer to the source D3DXVECTOR2 structure.

pM [in]

Type: **const D3DXMATRIX***

Pointer to the source **D3DXMATRIX** structure.

Return value

Type: **D3DXVECTOR2***

Pointer to a D3DXVECTOR2 structure that is the transformed vector.

Remarks

This function transforms the vector, pV (x, y, 0, 1), by the matrix, pM, projecting the result back into w=1.

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXVec2TransformCoord function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXVec2TransformCoordArray function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Transforms an array (x, y, 0, 1) by a given matrix, and projects the result back into w = 1.

Syntax

```C++
D3DXVECTOR2* D3DXVec2TransformCoordArray(
  _Inout_       D3DXVECTOR2 *pOut,
  _In_          UINT        OutStride,
  _In_    const D3DXVECTOR2 *pV,
  _In_          UINT        VStride,
  _In_    const D3DXMATRIX  *pM,
  _In_          UINT        n
);
```

Parameters

pOut [in, out]

Type: **D3DXVECTOR2***

Pointer to the **D3DXVECTOR2** that is the result of the operation.

OutStride [in]

Type: **UINT**

Stride between vectors in the output data stream.

pV [in]

Type: **const D3DXVECTOR2***

Pointer to the source D3DXVECTOR2 array.

VStride [in]

Type: **UINT**

Stride between vectors in the input data stream.

pM [in]

Type: **const D3DXMATRIX***

Pointer to the source **D3DXMATRIX** structure.

n [in]

Type: **UINT**

Number of elements in the array.

Return value

Type: **D3DXVECTOR2***

Pointer to a **D3DXVECTOR4** transformed array.

Remarks

This function transforms the array pV (x, y, 0, 1) by the matrix pM, projecting the result back into w = 1.

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXVec2TransformCoordArray function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXVec2TransformNormal function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Transforms the 2D vector normal by the given matrix.

Syntax

```
C++

D3DXVECTOR2* D3DXVec2TransformNormal(
  _Inout_       D3DXVECTOR2 *pOut,
  _In_    const D3DXVECTOR2 *pV,
  _In_    const D3DXMATRIX  *pM
);
```

Parameters

pOut [in, out]

Type: **D3DXVECTOR2***

Pointer to the **D3DXVECTOR2** that is the result of the operation.

pV [in]

Type: **const D3DXVECTOR2***

Pointer to the source D3DXVECTOR2 structure.

pM [in]

Type: **const D3DXMATRIX***

Pointer to the source **D3DXMATRIX** structure.

Return value

Type: **D3DXVECTOR2***

Pointer to a D3DXVECTOR2 structure that is the transformed vector.

Remarks

This function transforms the vector (pV->x, pV->y, 0, 0) by the matrix pointed to by pM.

If you want to transform a normal, the matrix you pass to this function should be the transpose of the inverse of the matrix you would use to transform a point.

The return value for this function is the same value returned in the pOut parameter. In this way, the **D3DXVec2TransformNormal** function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXVec2TransformNormalArray function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Transforms an array (x, y, 0, 0) by a given matrix.

Syntax

```C++
D3DXVECTOR2* D3DXVec2TransformNormalArray(
  _Inout_       D3DXVECTOR2 *pOut,
  _In_          UINT        OutStride,
  _In_    const D3DXVECTOR2 *pV,
  _In_          UINT        VStride,
  _In_    const D3DXMATRIX  *pM,
  _In_          UINT        n
);
```

Parameters

pOut [in, out]

Type: **D3DXVECTOR2***

Pointer to the **D3DXVECTOR2** that is the result of the operation.

OutStride [in]

Type: **UINT**

Stride between vectors in the output data stream.

pV [in]

Type: **const D3DXVECTOR2***

Pointer to the source D3DXVECTOR2 array.

VStride [in]

Type: **UINT**

Stride between vectors in the input data stream.

pM [in]

Type: **const D3DXMATRIX***

Pointer to the source **D3DXMATRIX** structure.

n [in]

Type: **UINT**

Number of elements in the array.

Return value

Type: **D3DXVECTOR2***

Pointer to a D3DXVECTOR2 structure that is the transformed array.

Remarks

This function transforms the vector (pV->x, pV->y, 0, 0) by the matrix pointed to by pM.

If you want to transform a normal, the matrix you pass to this function should be the transpose of the inverse of the matrix you would use to transform a point.

The return value for this function is the same value returned in the pOut parameter. In this way, the **D3DXVec2TransformNormalArray** function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h
Library	D3DX10.lib

See also

Math Functions

D3DXVec3BaryCentric function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Returns a point in Barycentric coordinates, using the specified 3D vectors.

Syntax

```C++
D3DXVECTOR3* D3DXVec3BaryCentric(
  _In_       D3DXVECTOR3 *pOut,
  _In_ const D3DXVECTOR3 *pV1,
  _In_ const D3DXVECTOR3 *pV2,
  _In_ const D3DXVECTOR3 *pV3,
  _In_       FLOAT       f,
  _In_       FLOAT       g
);
```

Parameters

pOut [in]

Type: **D3DXVECTOR3***

Pointer to the **D3DXVECTOR3** that is the result of the operation.

pV1 [in]

Type: **const D3DXVECTOR3***

Pointer to a source D3DXVECTOR3 structure.

pV2 [in]

Type: **const D3DXVECTOR3***

Pointer to a source D3DXVECTOR3 structure.

pV3 [in]

Type: **const** **D3DXVECTOR3***

Pointer to a source D3DXVECTOR3 structure.

f [in]

Type: **FLOAT**

Weighting factor. See Remarks.

g [in]

Type: **FLOAT**

Weighting factor. See Remarks.

Return value

Type: **D3DXVECTOR3***

Pointer to a D3DXVECTOR3 structure in Barycentric coordinates.

Remarks

The D3DXVec3BaryCentric function provides a way to understand points in and around a triangle, independent of where the triangle is actually located. This function returns the resulting point by using the following equation: V1 + f(V2-V1) + g(V3-V1).

Any point in the plane V1V2V3 can be represented by the Barycentric coordinate (f,g).The parameter f controls how much V2 gets weighted into the result and the parameter g controls how much V3 gets weighted into the result. Lastly, 1-f-g controls how much V1 gets weighted into the result.

Note the following relations.

- If (f>=0 &, & g>=0 &, & 1-f-g>=0), the point is inside the triangle V1V2V3.
- If (f==0 &, & g>=0 &, & 1-f-g>=0), the point is on the line V1V3.
- If (f>=0 &, & g==0 &, & 1-f-g>=0), the point is on the line V1V2.
- If (f>=0 &, & g>=0 &, & 1-f-g==0), the point is on the line V2V3.

Barycentric coordinates are a form of general coordinates. In this context, using Barycentric coordinates represents a change in coordinate systems. What holds true for Cartesian coordinates holds true for Barycentric coordinates.

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXVec3BaryCentric function can be used as a parameter for another function.

Barycentric coordinates define a point inside a triangle in terms of the triangle's vertices. For a more in-depth description of barycentric coordinates, see Mathworld's Barycentric Coordinates Description.

Requirements

Requirement	Value
Header	D3DX10Math.h

See also

Math Functions

D3DXVec3CatmullRom function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Performs a Catmull-Rom interpolation, using the specified 3D vectors.

Syntax

```C++
D3DXVECTOR3* D3DXVec3CatmullRom(
  _Inout_       D3DXVECTOR3 *pOut,
  _In_    const D3DXVECTOR3 *pV0,
  _In_    const D3DXVECTOR3 *pV1,
  _In_    const D3DXVECTOR3 *pV2,
  _In_    const D3DXVECTOR3 *pV3,
  _In_          FLOAT       s
);
```

Parameters

pOut [in, out]

Type: **D3DXVECTOR3***

Pointer to the **D3DXVECTOR3** that is the result of the operation.

pV0 [in]

Type: **const D3DXVECTOR3***

Pointer to a source D3DXVECTOR3 structure, a position vector.

pV1 [in]

Type: **const D3DXVECTOR3***

Pointer to a source D3DXVECTOR3 structure, a position vector.

pV2 [in]

Type: **const** D3DXVECTOR3*

Pointer to a source D3DXVECTOR3 structure, a position vector.

pV3 [in]

Type: **const** D3DXVECTOR3*

Pointer to a source D3DXVECTOR3 structure, a position vector.

s [in]

Type: FLOAT

Weighting factor. See Remarks.

Return value

Type: D3DXVECTOR3*

Pointer to a D3DXVECTOR3 structure that is the result of the Catmull-Rom interpolation.

Remarks

Given four points (p1, p2, p3, p4), find a function Q(s) such that:

```
Q(s) is a cubic function.
Q(s) interpolates between p2 and p3 as s ranges from 0 to 1.
Q(s) is parallel to the line joining p1 to p3 when s is 0.
Q(s) is parallel to the line joining p2 to p4 when s is 1.
```

The Catmull-Rom spline can be derived from the Hermite spline by setting:

```
v1 = p2
v2 = p3
t1 = (p3 - p1) / 2
t2 = (p4 - p2) / 2
```

where:

v1 is the contents of pV0.

v2 is the contents of pV1.

p3 is the contents of pV2.

p4 is the contents of pV3.

Using the Hermite spline equation:

```
Q(s) = (2s3 - 3s2 + 1)v1 + (-2s3 + 3s2)v2 + (s3 - 2s2 + s)t1 + (s3 - s2)t2
```

and substituting for v1, v2, t1, t2 yields:

```
Q(s) = (2s3 - 3s2 + 1)p2 + (-2s3 + 3s2)p3 + (s3 - 2s2 + s)(p3 - p1) / 2 +
(s3 - s2)(p4 - p2) / 2
```

This can be rearranged as:

```
Q(s) = [(-s3 + 2s2 - s)p1 + (3s3 - 5s2 + 2)p2 + (-3s3 + 4s2 + s)p3 + (s3 -
s2)p4] / 2
```

Requirements

Requirement	Value
Header	D3DX10Math.h

See also

Math Functions

D3DXVec3Hermite function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Performs a Hermite spline interpolation, using the specified 3D vectors.

Syntax

```C++
D3DXVECTOR3* D3DXVec3Hermite(
  _Inout_       D3DXVECTOR3 *pOut,
  _In_    const D3DXVECTOR3 *pV1,
  _In_    const D3DXVECTOR3 *pT1,
  _In_    const D3DXVECTOR3 *pV2,
  _In_    const D3DXVECTOR3 *pT2,
  _In_          FLOAT       s
);
```

Parameters

pOut [in, out]

Type: **D3DXVECTOR3***

Pointer to the **D3DXVECTOR3** that is the result of the operation.

pV1 [in]

Type: **const D3DXVECTOR3***

Pointer to a source D3DXVECTOR3 structure, a position vector.

pT1 [in]

Type: **const D3DXVECTOR3***

Pointer to a source D3DXVECTOR3 structure, a tangent vector.

pV2 [in]

Type: **const** D3DXVECTOR3*

Pointer to a source D3DXVECTOR3 structure, a position vector.

pT2 [in]

Type: **const** D3DXVECTOR3*

Pointer to a source D3DXVECTOR3 structure, a tangent vector.

s [in]

Type: FLOAT

Weighting factor. See Remarks.

Return value

Type: D3DXVECTOR3*

Pointer to a D3DXVECTOR3 structure that is the result of the Hermite spline interpolation.

Remarks

The **D3DXVec3Hermite** function interpolates from (positionA, tangentA) to (positionB, tangentB) using Hermite spline interpolation.

The spline interpolation is a generalization of the ease-in, ease-out spline. The ramp is a function of Q(s) with the following properties.

$Q(s) = As^3 + Bs^2 + Cs + D$ (and therefore, $Q'(s) = 3As^2 + 2Bs + C$)

a) Q(0) = v1, so Q'(0) = t1

b) Q(1) = v2, so Q'(1) = t2

v1 is the contents of pV1, v2 in the contents of pV2, t1 is the contents of pT1, and t2 is the contents of pT2.

These properties are used to solve for A, B, C, D.

```
D = v1  (from a)
C = t1  (from a)
3A + 2B = t2 - t1 (substituting for C)
A + B = v2 - v1 - t1 (substituting for C and D)
```

Plug in the solutions for A,B,C and D to generate Q(s).

```
A = 2v1 - 2v2 + t2 + t1
B = 3v2 - 3v1 - 2t1 - t2
C = t1
D = v1
```

This yields:

$Q(s) = (2v1 - 2v2 + t2 + t1)s^3 + (3v2 - 3v1 - 2t1 - t2)s^2 + t1s + v1$

Which can be rearranged as:

$Q(s) = (2s^3 - 3s^2 + 1)v1 + (-2s^3 + 3s^2)v2 + (s^3 - 2s^2 + s)t1 + (s^3 - s^2)t2$

Hermite splines are useful for controlling animation because the curve runs through all the control points. Also, because the position and tangent are explicitly specified at the ends of each segment, it is easy to create a C2 continuous curve as long as you make sure that your starting position and tangent match the ending values of the last segment.

The return value for this function is the same value returned in the pOut parameter. In this way, the **D3DXVec3Hermite** function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h

See also

Math Functions

D3DXVec3Normalize function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Returns the normalized version of a 3D vector.

Syntax

```C++
D3DXVECTOR3* D3DXVec3Normalize(
  _Inout_       D3DXVECTOR3 *pOut,
  _In_    const D3DXVECTOR3 *pV
);
```

Parameters

pOut [in, out]

Type: **D3DXVECTOR3***

Pointer to the **D3DXVECTOR3** that is the result of the operation.

pV [in]

Type: **const D3DXVECTOR3***

Pointer to the source D3DXVECTOR3 structure.

Return value

Type: **D3DXVECTOR3***

Pointer to a D3DXVECTOR3 structure that is the normalized version of the specified vector.

Remarks

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXVec3Normalize function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h

See also

Math Functions

D3DXVec3Project function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Projects a 3D vector from object space into screen space.

Syntax

```C++
D3DXVECTOR3* D3DXVec3Project(
  _Inout_       D3DXVECTOR3    *pOut,
  _In_    const D3DXVECTOR3    *pV,
  _In_    const D3D10_VIEWPORT *pViewport,
  _In_    const D3DXMATRIX     *pProjection,
  _In_    const D3DXMATRIX     *pView,
  _In_    const D3DXMATRIX     *pWorld
);
```

Parameters

pOut [in, out]

Type: **D3DXVECTOR3***

Pointer to the **D3DXVECTOR3** that is the result of the operation.

pV [in]

Type: **const D3DXVECTOR3***

Pointer to the source D3DXVECTOR3 structure.

pViewport [in]

Type: **const D3D10_VIEWPORT***

Pointer to a **D3D10_VIEWPORT**, representing the viewport.

pProjection [in]

Type: **const D3DXMATRIX***

Pointer to a **D3DXMATRIX** structure, representing the projection matrix.

pView [in]

Type: **const D3DXMATRIX***

Pointer to a D3DXMATRIX structure, representing the view matrix.

pWorld [in]

Type: **const D3DXMATRIX***

Pointer to a D3DXMATRIX structure, representing the world matrix.

Return value

Type: **D3DXVECTOR3***

Pointer to a D3DXVECTOR3 structure that is the vector projected from object space to screen space.

Remarks

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXVec3Project function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h

See also

Math Functions

D3DXVec3ProjectArray function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Projects an array (x, y, z, 0) from object space into screen space.

Syntax

```C++
D3DXVECTOR3* D3DXVec3ProjectArray(
  _Inout_       D3DXVECTOR3    *pOut,
  _In_          UINT           OutStride,
  _In_    const D3DXVECTOR3    *pV,
  _In_          UINT           VStride,
  _In_    const D3D10_VIEWPORT *pViewport,
  _In_    const D3DXMATRIX     *pProjection,
  _In_    const D3DXMATRIX     *pView,
  _In_    const D3DXMATRIX     *pWorld,
  _In_          UINT           n
);
```

Parameters

pOut [in, out]

Type: **D3DXVECTOR3***

Pointer to the **D3DXVECTOR3** that is the result of the operation.

OutStride [in]

Type: **UINT**

Stride between vectors in the output data stream.

pV [in]

Type: **const** **D3DXVECTOR3***

Pointer to the source D3DXVECTOR3 structure.

VStride [in]

Type: **UINT**

Stride between vectors in the input data stream.

pViewport [in]

Type: **const** **D3D10_VIEWPORT***

Pointer to a **D3D10_VIEWPORT**, representing the viewport.

pProjection [in]

Type: **const** **D3DXMATRIX***

Pointer to a **D3DXMATRIX** structure, representing the projection matrix.

pView [in]

Type: **const** **D3DXMATRIX***

Pointer to a D3DXMATRIX structure, representing the view matrix.

pWorld [in]

Type: **const** **D3DXMATRIX***

Pointer to a D3DXMATRIX structure, representing the world matrix.

n [in]

Type: **UINT**

Number of elements in the array.

Return value

Type: **D3DXVECTOR3***

Pointer to a D3DXVECTOR3 structure that is the array projected from object space to screen space.

Remarks

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXVec3ProjectArray function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h

See also

Math Functions

D3DXVec3Transform function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Transforms vector (x, y, z, 1) by a given matrix.

Syntax

```C++
D3DXVECTOR4* D3DXVec3Transform(
  _Inout_       D3DXVECTOR4 *pOut,
  _In_    const D3DXVECTOR3 *pV,
  _In_    const D3DXMATRIX  *pM
);
```

Parameters

pOut [in, out]

Type: **D3DXVECTOR4***

Pointer to the **D3DXVECTOR4** structure that is the result of the operation.

pV [in]

Type: **const D3DXVECTOR3***

Pointer to the source **D3DXVECTOR3**.

pM [in]

Type: **const D3DXMATRIX***

Pointer to the source **D3DXMATRIX** structure.

Return value

Type: **D3DXVECTOR4***

Pointer to a D3DXVECTOR4 structure that is the transformed vector.

Remarks

This function transforms the vector, pV (x, y, z, 1), by the matrix pM.

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXVec3Transform function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h

See also

Math Functions

D3DXVec3TransformArray function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Transforms an array (x, y, z, 1) by a given matrix.

Syntax

```C++
D3DXVECTOR4* D3DXVec3TransformArray(
  _Inout_       D3DXVECTOR4 *pOut,
  _In_          UINT        OutStride,
  _In_    const D3DXVECTOR3 *pV,
  _In_          UINT        VStride,
  _In_    const D3DXMATRIX  *pM,
  _In_          UINT        n
);
```

Parameters

pOut [in, out]

Type: **D3DXVECTOR4***

Pointer to the **D3DXVECTOR4** array that is the result of the operation.

OutStride [in]

Type: **UINT**

Stride between vectors in the output data stream.

pV [in]

Type: **const D3DXVECTOR3***

Pointer to the source D3DXVECTOR3 array.

VStride [in]

Type: UINT

Stride between vectors in the input data stream.

pM [in]

Type: **const** D3DXMATRIX*

Pointer to the source D3DXMATRIX structure.

n [in]

Type: UINT

Number of elements in the array.

Return value

Type: D3DXVECTOR4*

Pointer to a D3DXVECTOR4 transformed array.

Remarks

This function transforms the array pV (x, y, z, 1) by the matrix pM.

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXVec3TransformArray function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h

See also

Math Functions

D3DXVec3TransformCoord function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Transforms a 3D vector by a given matrix, projecting the result back into w = 1.

Syntax

```C++
D3DXVECTOR3* D3DXVec3TransformCoord(
  _Inout_       D3DXVECTOR3 *pOut,
  _In_    const D3DXVECTOR3 *pV,
  _In_    const D3DXMATRIX  *pM
);
```

Parameters

pOut [in, out]

Type: **D3DXVECTOR3***

Pointer to the **D3DXVECTOR3** that is the result of the operation.

pV [in]

Type: **const D3DXVECTOR3***

Pointer to the source D3DXVECTOR3 structure.

pM [in]

Type: **const D3DXMATRIX***

Pointer to the source **D3DXMATRIX** structure.

Return value

Type: **D3DXVECTOR3***

Pointer to a D3DXVECTOR3 structure that is the transformed vector.

Remarks

This function transforms the vector, pV (x, y, z, 1), by the matrix, pM, projecting the result back into w=1.

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXVec3TransformCoord function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h

See also

Math Functions

D3DXVec3TransformCoordArray function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Transforms an array (x, y, z, 1) by a given matrix, and projects the result back into w = 1.

Syntax

```C++
D3DXVECTOR3* D3DXVec3TransformCoordArray(
  _Inout_       D3DXVECTOR3 *pOut,
  _In_          UINT        OutStride,
  _In_    const D3DXVECTOR3 *pV,
  _In_          UINT        VStride,
  _In_    const D3DXMATRIX  *pM,
  _In_          UINT        n
);
```

Parameters

pOut [in, out]

Type: **D3DXVECTOR3***

Pointer to the **D3DXVECTOR3** that is the result of the operation.

OutStride [in]

Type: **UINT**

Stride between vectors in the output data stream.

pV [in]

Type: **const D3DXVECTOR3***

Pointer to the source D3DXVECTOR3 array.

VStride [in]

Type: **UINT**

Stride between vectors in the input data stream.

pM [in]

Type: **const D3DXMATRIX***

Pointer to the source **D3DXMATRIX** structure.

n [in]

Type: **UINT**

Number of elements in the array.

Return value

Type: **D3DXVECTOR3***

Pointer to a D3DXVECTOR3 structure that is the transformed array.

Remarks

This function transforms the array pV (x, y, z, 1) by the matrix pM, projecting the result back into w = 1.

The return value for this function is the same value returned in the pOut parameter. In this way, the **D3DXVec3TransformCoord** function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h

See also

Math Functions

D3DXVec3TransformNormal function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Transforms the 3D vector normal by the given matrix.

Syntax

```C++
D3DXVECTOR3* D3DXVec3TransformNormal(
  _Inout_       D3DXVECTOR3 *pOut,
  _In_    const D3DXVECTOR3 *pV,
  _In_    const D3DXMATRIX  *pM
);
```

Parameters

pOut [in, out]

Type: **D3DXVECTOR3***

Pointer to the **D3DXVECTOR3** that is the result of the operation.

pV [in]

Type: **const D3DXVECTOR3***

Pointer to the source D3DXVECTOR3 structure.

pM [in]

Type: **const D3DXMATRIX***

Pointer to the source **D3DXMATRIX** structure.

Return value

Type: **D3DXVECTOR3***

Pointer to a D3DXVECTOR3 structure that is the transformed vector.

Remarks

This function transforms the vector (pV->x, pV->y, pV->z, 0) by the matrix pointed to by pM.

If you want to transform a normal, the matrix you pass to this function should be the transpose of the inverse of the matrix you would use to transform a point.

The return value for this function is the same value returned in the pOut parameter. In this way, the **D3DXVec3TransformNormal** function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h

See also

Math Functions

D3DXVec3TransformNormalArray function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Transforms an array (x, y, z, 0) by a given matrix.

Syntax

```C++
D3DXVECTOR3* D3DXVec3TransformNormalArray(
  _Inout_       D3DXVECTOR3 *pOut,
  _In_          UINT        OutStride,
  _In_    const D3DXVECTOR3 *pV,
  _In_          UINT        VStride,
  _In_    const D3DXMATRIX  *pM,
  _In_          UINT        n
);
```

Parameters

pOut [in, out]

Type: **D3DXVECTOR3***

Pointer to the **D3DXVECTOR3** array that is the result of the operation.

OutStride [in]

Type: **UINT**

Stride between vectors in the output data stream.

pV [in]

Type: **const D3DXVECTOR3***

Pointer to the source D3DXVECTOR3 array.

VStride [in]

Type: **UINT**

Stride between vectors in the input data stream.

pM [in]

Type: **const D3DXMATRIX***

Pointer to the source **D3DXMATRIX** structure.

n [in]

Type: **UINT**

Number of elements in the array.

Return value

Type: **D3DXVECTOR3***

Pointer to a D3DXVECTOR3 array that is the transformed array.

Remarks

This function transforms the vector (pV->x, pV->y, pV->z, 0) by the matrix pointed to by pM.

If you want to transform a normal, the matrix you pass to this function should be the transpose of the inverse of the matrix you would use to transform a point.

The return value for this function is the same value returned in the pOut parameter. In this way, the **D3DXVec3TransformNormalArray** function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h

See also

Math Functions

D3DXVec3Unproject function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Projects a vector from screen space into object space.

Syntax

```C++
D3DXVECTOR3* D3DXVec3Unproject(
  _Inout_       D3DXVECTOR3    *pOut,
  _In_    const D3DXVECTOR3    *pV,
  _In_    const D3D10_VIEWPORT *pViewport,
  _In_    const D3DXMATRIX     *pProjection,
  _In_    const D3DXMATRIX     *pView,
  _In_    const D3DXMATRIX     *pWorld
);
```

Parameters

pOut [in, out]

Type: **D3DXVECTOR3***

Pointer to the **D3DXVECTOR3** that is the result of the operation.

pV [in]

Type: **const D3DXVECTOR3***

Pointer to the source D3DXVECTOR3 structure.

pViewport [in]

Type: **const D3D10_VIEWPORT***

Pointer to a **D3D10_VIEWPORT**, representing the viewport.

pProjection [in]

Type: **const D3DXMATRIX***

Pointer to a **D3DXMATRIX** structure, representing the projection matrix.

pView [in]

Type: **const D3DXMATRIX***

Pointer to a D3DXMATRIX structure, representing the view matrix.

pWorld [in]

Type: **const D3DXMATRIX***

Pointer to a D3DXMATRIX structure, representing the world matrix.

Return value

Type: **D3DXVECTOR3***

Pointer to a D3DXVECTOR3 structure that is the vector projected from screen space to object space.

Remarks

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXVec3Unproject function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h

See also

Math Functions

D3DXVec3UnprojectArray function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Projects an array (x, y, z, 0) from screen space into object space.

Syntax

```C++
D3DXVECTOR3* D3DXVec3UnprojectArray(
  _Inout_       D3DXVECTOR3    *pOut,
  _In_          UINT           OutStride,
  _In_    const D3DXVECTOR3    *pV,
  _In_          UINT           VStride,
  _In_    const D3D10_VIEWPORT *pViewport,
  _In_    const D3DXMATRIX     *pProjection,
  _In_    const D3DXMATRIX     *pView,
  _In_    const D3DXMATRIX     *pWorld,
  _In_          UINT           n
);
```

Parameters

pOut [in, out]

Type: **D3DXVECTOR3***

Pointer to the **D3DXVECTOR3** that is the result of the operation.

OutStride [in]

Type: **UINT**

Stride between vectors in the output data stream.

pV [in]

Type: **const** **D3DXVECTOR3***

Pointer to the source D3DXVECTOR3 structure.

VStride [in]

Type: **UINT**

Stride between vectors in the input data stream.

pViewport [in]

Type: **const** **D3D10_VIEWPORT***

Pointer to a **D3D10_VIEWPORT**, representing the viewport.

pProjection [in]

Type: **const** **D3DXMATRIX***

Pointer to a **D3DXMATRIX** structure, representing the projection matrix.

pView [in]

Type: **const** **D3DXMATRIX***

Pointer to a D3DXMATRIX structure, representing the view matrix.

pWorld [in]

Type: **const** **D3DXMATRIX***

Pointer to a D3DXMATRIX structure, representing the world matrix.

n [in]

Type: **UINT**

Number of elements in the array.

Return value

Type: **D3DXVECTOR3***

Pointer to a D3DXVECTOR3 structure that is the array projected from screen space to object space.

Remarks

The return value for this function is the same value returned in the pOut parameter. In this way, the **D3DXVec3Unproject** function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h

See also

Math Functions

D3DXVec4BaryCentric function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Returns a point in Barycentric coordinates, using the specified 4D vectors.

Syntax

```C++
D3DXVECTOR4* D3DXVec4BaryCentric(
  _In_       D3DXVECTOR4 *pOut,
  _In_ const D3DXVECTOR4 *pV1,
  _In_ const D3DXVECTOR4 *pV2,
  _In_ const D3DXVECTOR4 *pV3,
  _In_       FLOAT       f,
  _In_       FLOAT       g
);
```

Parameters

pOut [in]

Type: **D3DXVECTOR4***

Pointer to the **D3DXVECTOR4** that is the result of the operation.

pV1 [in]

Type: **const D3DXVECTOR4***

Pointer to a source D3DXVECTOR4 structure.

pV2 [in]

Type: **const D3DXVECTOR4***

Pointer to a source D3DXVECTOR4 structure.

pV3 [in]

Type: **const** **D3DXVECTOR4***

Pointer to a source D3DXVECTOR4 structure.

f [in]

Type: **FLOAT**

Weighting factor. See Remarks.

g [in]

Type: **FLOAT**

Weighting factor. See Remarks.

Return value

Type: **D3DXVECTOR4***

Pointer to a D3DXVECTOR4 structure in Barycentric coordinates.

Remarks

The D3DXVec4BaryCentric function provides a way to understand points in and around a triangle, independent of where the triangle is actually located. This function returns the resulting point by using the following equation: V1 + f(V2-V1) + g(V3-V1).

Any point in the plane V1V2V3 can be represented by the Barycentric coordinate (f,g).The parameter f controls how much V2 gets weighted into the result and the parameter g controls how much V3 gets weighted into the result. Lastly, 1-f-g controls how much V1 gets weighted into the result.

Note the following relations.

- If (f>=0 &, & g>=0 &, & 1-f-g>=0), the point is inside the triangle V1V2V3.
- If (f==0 &, & g>=0 &, & 1-f-g>=0), the point is on the line V1V3.
- If (f>=0 &, & g==0 &, & 1-f-g>=0), the point is on the line V1V2.
- If (f>=0 &, & g>=0 &, & 1-f-g==0), the point is on the line V2V3.

Barycentric coordinates are a form of general coordinates. In this context, using Barycentric coordinates represents a change in coordinate systems. What holds true for Cartesian coordinates holds true for Barycentric coordinates.

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXVec4BaryCentric function can be used as a parameter for another function.

Barycentric coordinates define a point inside a triangle in terms of the triangle's vertices. For a more in-depth description of barycentric coordinates, see Mathworld's Barycentric Coordinates Description.

Requirements

Requirement	Value
Header	D3DX10Math.h

See also

Math Functions

D3DXVec4CatmullRom function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Performs a Catmull-Rom interpolation, using the specified 4D vectors.

Syntax

```C++
D3DXVECTOR4* D3DXVec4CatmullRom(
  _Inout_       D3DXVECTOR4 *pOut,
  _In_    const D3DXVECTOR4 *pV0,
  _In_    const D3DXVECTOR4 *pV1,
  _In_    const D3DXVECTOR4 *pV2,
  _In_    const D3DXVECTOR4 *pV3,
  _In_          FLOAT       s
);
```

Parameters

pOut [in, out]

Type: **D3DXVECTOR4***

Pointer to the **D3DXVECTOR4** that is the result of the operation.

pV0 [in]

Type: **const D3DXVECTOR4***

Pointer to a source D3DXVECTOR4 structure, a position vector.

pV1 [in]

Type: **const D3DXVECTOR4***

Pointer to a source D3DXVECTOR4 structure, a position vector.

pV2 [in]

Type: **const** D3DXVECTOR4*

Pointer to a source D3DXVECTOR4 structure, a position vector.

pV3 [in]

Type: **const** D3DXVECTOR4*

Pointer to a source D3DXVECTOR4 structure, a position vector.

s [in]

Type: FLOAT

Weighting factor. See Remarks.

Return value

Type: D3DXVECTOR4*

Pointer to a D3DXVECTOR4 structure that is the result of the Catmull-Rom interpolation.

Remarks

Given four points (p1, p2, p3, p4), find a function Q(s) such that:

```
Q(s) is a cubic function.
Q(s) interpolates between p2 and p3 as s ranges from 0 to 1.
Q(s) is parallel to the line joining p1 to p3 when s is 0.
Q(s) is parallel to the line joining p2 to p4 when s is 1.
```

The Catmull-Rom spline can be derived from the Hermite spline by setting:

```
v1 = p2
v2 = p3
t1 = (p3 - p1) / 2
t2 = (p4 - p2) / 2
```

where:

v1 is the contents of pV0.

v2 in the contents of pV1.

p3 is the contents of pV2.

p4 is the contents of pV3.

Using the Hermite spline equation:

```
Q(s) = (2s3 - 3s2 + 1)v1 + (-2s3 + 3s2)v2 + (s3 - 2s2 + s)t1 + (s3 - s2)t2
```

and substituting for v1, v2, t1, t2 yields:

```
Q(s) = (2s3 - 3s2 + 1)p2 + (-2s3 + 3s2)p3 + (s3 - 2s2 + s)(p3 - p1) / 2 +
(s3 - s2)(p4 - p2) / 2
```

This can be rearranged as:

```
Q(s) = [(-s3 + 2s2 - s)p1 + (3s3 - 5s2 + 2)p2 + (-3s3 + 4s2 + s)p3 + (s3 -
s2)p4] / 2
```

Requirements

Requirement	Value
Header	D3DX10Math.h

See also

Math Functions

D3DXVec4Cross function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Determines the cross-product in four dimensions.

Syntax

```C++
D3DXVECTOR4* D3DXVec4Cross(
  _Inout_       D3DXVECTOR4 *pOut,
  _In_    const D3DXVECTOR4 *pV1,
  _In_    const D3DXVECTOR4 *pV2,
  _In_    const D3DXVECTOR4 *pV3
);
```

Parameters

pOut [in, out]

Type: **D3DXVECTOR4***

Pointer to the **D3DXVECTOR4** that is the result of the operation.

pV1 [in]

Type: **const D3DXVECTOR4***

Pointer to a source D3DXVECTOR4 structure.

pV2 [in]

Type: **const D3DXVECTOR4***

Pointer to a source D3DXVECTOR4 structure.

pV3 [in]

Type: **const** **D3DXVECTOR4***

Pointer to a source D3DXVECTOR4 structure.

Return value

Type: **D3DXVECTOR4***

Pointer to a D3DXVECTOR4 structure that is the cross product.

Remarks

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXVec4Cross function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h

See also

Math Functions

D3DXVec4Hermite function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Performs a Hermite spline interpolation, using the specified 4D vectors.

Syntax

```C++
D3DXVECTOR4* D3DXVec4Hermite(
  _Inout_       D3DXVECTOR4 *pOut,
  _In_    const D3DXVECTOR4 *pV1,
  _In_    const D3DXVECTOR4 *pT1,
  _In_    const D3DXVECTOR4 *pV2,
  _In_    const D3DXVECTOR4 *pT2,
  _In_          FLOAT       s
);
```

Parameters

pOut [in, out]

Type: **D3DXVECTOR4***

Pointer to the **D3DXVECTOR4** that is the result of the operation.

pV1 [in]

Type: **const D3DXVECTOR4***

Pointer to a source D3DXVECTOR4 structure, a position vector.

pT1 [in]

Type: **const D3DXVECTOR4***

Pointer to a source D3DXVECTOR4 structure, a tangent vector.

pV2 [in]

Type: **const** **D3DXVECTOR4***

Pointer to a source D3DXVECTOR4 structure, a position vector.

pT2 [in]

Type: **const** **D3DXVECTOR4***

Pointer to a source D3DXVECTOR4 structure, a tangent vector.

s [in]

Type: **FLOAT**

Weighting factor. See Remarks.

Return value

Type: **D3DXVECTOR4***

Pointer to a D3DXVECTOR4 structure that is the result of the Hermite spline interpolation.

Remarks

The **D3DXVec4Hermite** function interpolates from (positionA, tangentA) to (positionB, tangentB) using Hermite spline interpolation.

The spline interpolation is a generalization of the ease-in, ease-out spline. The ramp is a function of Q(s) with the following properties.

$Q(s) = As^3 + Bs^2 + Cs + D$ (and therefore, $Q'(s) = 3As^2 + 2Bs + C$)

a) Q(0) = v1, so Q'(0) = t1

b) Q(1) = v2, so Q'(1) = t2

v1 is the contents of pV1, v2 in the contents of pV2, t1 is the contents of pT1, and t2 is the contents of pT2.

These properties are used to solve for A, B, C, D.

```
D = v1  (from a)
C = t1  (from a)
3A + 2B = t2 - t-1 (substituting for C)
A + B = v2 - v1 - t1 (substituting for C and D)
```

Plug in the solutions for A,B,C and D to generate Q(s).

```
A = 2v1 - 2v2 + t2 + t1
B = 3v2 - 3v1 - 2t1 - t2
C = t1
D = v1
```

This yields:

$Q(s) = (2v1 - 2v2 + t2 + t1)s^3 + (3v2 - 3v1 - 2t1 - t2)s^2 + t1s + v1$

Which can be rearranged as:

$Q(s) = (2s^3 - 3s^2 + 1)v1 + (-2s^3 + 3s^2)v2 + (s^3 - 2s^2 + s)t1 + (s^3 - s^2)t2$

Hermite splines are useful for controlling animation because the curve runs through all the control points. Also, because the position and tangent are explicitly specified at the ends of each segment, it is easy to create a C2 continuous curve as long as you make sure that your starting position and tangent match the ending values of the last segment.

The return value for this function is the same value returned in the pOut parameter. In this way, the **D3DXVec4Hermite** function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h

See also

Math Functions

D3DXVec4Normalize function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Returns the normalized version of a 4D vector.

Syntax

```C++
D3DXVECTOR4* D3DXVec4Normalize(
  _Inout_       D3DXVECTOR4 *pOut,
  _In_    const D3DXVECTOR4 *pV
);
```

Parameters

pOut [in, out]

Type: **D3DXVECTOR4***

Pointer to the **D3DXVECTOR4** that is the result of the operation.

pV [in]

Type: **const D3DXVECTOR4***

Pointer to the source D3DXVECTOR4 structure.

Return value

Type: **D3DXVECTOR4***

Pointer to a D3DXVECTOR4 structure that is the normalized version of the vector.

Remarks

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXVec4Normalize function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h

See also

Math Functions

D3DXVec4Transform function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Transforms a 4D vector by a given matrix.

Syntax

```C++
D3DXVECTOR4* D3DXVec4Transform(
  _Inout_       D3DXVECTOR4 *pOut,
  _In_    const D3DXVECTOR4 *pV,
  _In_    const D3DXMATRIX  *pM
);
```

Parameters

pOut [in, out]

Type: **D3DXVECTOR4***

Pointer to the **D3DXVECTOR4** that is the result of the operation.

pV [in]

Type: **const D3DXVECTOR4***

Pointer to the source D3DXVECTOR4 structure.

pM [in]

Type: **const D3DXMATRIX***

Pointer to the source **D3DXMATRIX** structure.

Return value

Type: **D3DXVECTOR4***

Pointer to a D3DXVECTOR4 structure that is the transformed 4D vector.

Remarks

The return value for this function is the same value returned in the pOut parameter. In this way, the D3DXVec4Transform function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h

See also

Math Functions

D3DXVec4TransformArray function (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Transforms an array (x, y, z, w) by a given matrix.

Syntax

C++

```
D3DXVECTOR4* D3DXVec4TransformArray(
  _Inout_       D3DXVECTOR4 *pOut,
  _In_          UINT        OutStride,
  _In_    const D3DXVECTOR4 *pV,
  _In_          UINT        VStride,
  _In_    const D3DXMATRIX  *pM,
  _In_          UINT        n
);
```

Parameters

pOut [in, out]

Type: **D3DXVECTOR4***

Pointer to the **D3DXVECTOR4** array that is the result of the operation.

OutStride [in]

Type: **UINT**

Stride between vectors in the output data stream.

pV [in]

Type: **const D3DXVECTOR4***

Pointer to the source D3DXVECTOR4 array.

VStride [in]

Type: **UINT**

Stride between vectors in the input data stream.

pM [in]

Type: **const D3DXMATRIX***

Pointer to the source **D3DXMATRIX** structure.

n [in]

Type: **UINT**

Number of elements in the array.

Return value

Type: **D3DXVECTOR4***

Pointer to a **D3DXVECTOR4** structure that is the transformed array.

Remarks

This function transforms the array pV (x, y, z, w) by the matrix pM.

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec4TransformArray** function can be used as a parameter for another function.

Requirements

Requirement	Value
Header	D3DX10Math.h

See also

Math Functions

Mesh Functions (Direct3D 10 Graphics)

Article • 06/18/2021 • 2 minutes to read

D3DX supplies the following mesh functions.

- **D3DX10CreateMesh**
- **D3DX10CreateSkinInfo**
- **D3DXboxBoundProbe**
- **D3DXComputeBoundingBox**
- **D3DXComputeBoundingSphere**
- **D3DXIntersectTri**
- **D3DXSphereBoundProbe**

Related topics

D3DX Functions

D3DX10CreateMesh function

Article • 04/27/2021 • 2 minutes to read

Creates a mesh object using a declarator.

Syntax

```C++
HRESULT D3DX10CreateMesh(
  _In_        ID3D10Device             *pDevice,
  _In_  const D3D10_INPUT_ELEMENT_DESC *pDeclaration,
  _In_        UINT                     DeclCount,
  _In_        LPCSTR                   pPositionSemantic,
  _In_        UINT                     VertexCount,
  _In_        UINT                     FaceCount,
  _In_        UINT                     Options,
  _Out_       ID3DX10Mesh              **ppMesh
);
```

Parameters

pDevice [in]

Type: **ID3D10Device***

Pointer to an **ID3D10Device Interface**, the device object to be associated with the mesh.

pDeclaration [in]

Type: **const D3D10_INPUT_ELEMENT_DESC***

Array of **D3D10_INPUT_ELEMENT_DESC** elements, describing the vertex format for the returned mesh. This parameter must map directly to a flexible vertex format (FVF).

DeclCount [in]

Type: **UINT**

The number of elements in pDeclaration.

pPositionSemantic [in]

Type: **LPCSTR**

Semantic that identifies which part of the vertex declaration contains position information.

VertexCount [in]

Type: **UINT**

Number of vertices for the mesh. This parameter must be greater than 0.

FaceCount [in]

Type: **UINT**

Number of faces for the mesh. The valid range for this number is greater than 0, and one less than the maximum DWORD (typically 65534), because the last index is reserved.

Options [in]

Type: **UINT**

Combination of one or more flags from the **D3DX10_MESH**, specifying options for the mesh.

ppMesh [out]

Type: **ID3DX10Mesh****

Address of a pointer to an **ID3DX10Mesh Interface**, representing the created mesh object.

Return value

Type: **HRESULT**

If the function succeeds, the return value is D3D_OK. If the function fails, the return value can be one of the following: D3DERR_INVALIDCALL, E_OUTOFMEMORY.

Requirements

Requirement	Value
Header	D3DX10Mesh.h
Library	D3DX10.lib

See also

Mesh Functions

D3DX Functions

D3DX10CreateSkinInfo function

Article • 04/27/2021 • 2 minutes to read

Creates an empty skin mesh object using a declarator.

Syntax

```
C++

HRESULT D3DX10CreateSkinInfo(
  _Out_ LPD3DX10SKININFO *ppSkinInfo
);
```

Parameters

ppSkinInfo [out]

Type: **LPD3DX10SKININFO***

Address of a pointer to an **ID3DX10SkinInfo Interface**, representing the created skin mesh object.

Return value

Type: **HRESULT**

If the function succeeds, the return value is D3D_OK. If the function fails, the return value can be: E_OUTOFMEMORY.

Remarks

Use the **ID3DX10SkinInfo::SetBoneInfluence** to populate the empty skin mesh object returned by this method.

Requirements

Requirement	Value
Header	D3DX10Mesh.h

Requirement	Value
Library	D3DX10.lib

See also

Mesh Functions

D3DX Functions

D3DXBoxBoundProbe function (D3DX10math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Determines whether a ray intersects the volume of a box's bounding box.

Syntax

```C++
BOOL D3DXBoxBoundProbe(
  _In_ const D3DXVECTOR3 *pMin,
  _In_ const D3DXVECTOR3 *pMax,
  _In_ const D3DXVECTOR3 *pRayPosition,
  _In_ const D3DXVECTOR3 *pRayDirection
);
```

Parameters

pMin [in]

Type: **const** D3DXVECTOR3*

Pointer to a D3DXVECTOR3, describing the lower-left corner of the bounding box. See Remarks.

pMax [in]

Type: **const** D3DXVECTOR3*

Pointer to a D3DXVECTOR3 structure, describing the upper-right corner of the bounding box. See Remarks.

pRayPosition [in]

Type: **const** D3DXVECTOR3*

Pointer to a D3DXVECTOR3 structure, specifying the origin coordinate of the ray.

pRayDirection [in]

Type: **const** D3DXVECTOR3*

Pointer to a D3DXVECTOR3 structure, specifying the direction of the ray. This vector should not be (0,0,0) but does not need to be normalized.

Return value

Type: BOOL

Returns **TRUE** if the ray intersects the volume of the box's bounding box. Otherwise, returns **FALSE**.

Remarks

D3DXBoxBoundProbe determines if the ray intersects the volume of the box's bounding box, not just the surface of the box.

The values passed to **D3DXBoxBoundProbe** are xmin, xmax, ymin, ymax, zmin, and zmax. Thus, the following defines the corners of the bounding box.

```
xmax, ymax, zmax
xmax, ymax, zmin
xmax, ymin, zmax
xmax, ymin, zmin
xmin, ymax, zmax
xmin, ymax, zmin
xmin, ymin, zmax
xmin, ymin, zmin
```

The depth of the bounding box in the z direction is zmax - zmin, in the y direction is ymax - ymin, and in the x direction is xmax - xmin. For example, with the following minimum and maximum vectors, min (-1, -1, -1) and max (1, 1, 1), the bounding box is defined in the following manner.

```
 1,  1,  1
 1,  1, -1
 1, -1,  1
```

```
 1, -1, -1
-1,  1,  1
-1,  1, -1
-1, -1,  1
-1, -1, -l
```

Requirements

Requirement	Value
Header	D3DX10math.h
Library	D3DX10.lib

See also

Mesh Functions

D3DXComputeBoundingBox function (D3DX10math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Computes a coordinate-axis oriented bounding box.

Syntax

```C++
HRESULT D3DXComputeBoundingBox(
  _In_  const D3DXVECTOR3 *pFirstPosition,
  _In_        DWORD       NumVertices,
  _In_        DWORD       dwStride,
  _Out_       D3DXVECTOR3 *pMin,
  _Out_       D3DXVECTOR3 *pMax
);
```

Parameters

pFirstPosition [in]

Type: **const** D3DXVECTOR3*

Pointer to the first position.

NumVertices [in]

Type: DWORD

Number of vertices.

dwStride [in]

Type: DWORD

Count or number of bytes between vertices.

pMin [out]

Type: **D3DXVECTOR3***

Pointer to a **D3DXVECTOR3** structure, describing the returned lower-left corner of the bounding box.

pMax [out]

Type: **D3DXVECTOR3***

Pointer to a **D3DXVECTOR3** structure, describing the returned upper-right corner of the bounding box.

Return value

Type: **HRESULT**

If the function succeeds, the return value is D3D_OK. If the function fails, the return value can be one of the following: D3DERR_INVALIDCALL.

Requirements

Requirement	Value
Header	D3DX10math.h
Library	D3DX10.lib

See also

Mesh Functions

D3DXComputeBoundingSphere function (D3DX10math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Computes a bounding sphere for the mesh.

Syntax

```C++
HRESULT D3DXComputeBoundingSphere(
  _In_ const D3DXVECTOR3 *pFirstPosition,
  _In_       DWORD       NumVertices,
  _In_       DWORD       dwStride,
  _In_       D3DXVECTOR3 *pCenter,
  _In_       FLOAT       *pRadius
);
```

Parameters

pFirstPosition [in]

Type: **const** D3DXVECTOR3*

Pointer to first position.

NumVertices [in]

Type: DWORD

Number of vertices.

dwStride [in]

Type: DWORD

Number of bytes between position vectors.

pCenter [in]

Type: **D3DXVECTOR3***

D3DXVECTOR3 structure, defining the coordinate center of the returned bounding sphere.

pRadius [in]

Type: **FLOAT***

Radius of the returned bounding sphere.

Return value

Type: **HRESULT**

If the function succeeds, the return value is D3D_OK. If the function fails, the return value can be one of the following: D3DERR_INVALIDCALL.

Requirements

Requirement	Value
Header	D3DX10math.h
Library	D3DX10.lib

See also

Mesh Functions

D3DXIntersectTri function (D3DX10math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX utility library is deprecated. We recommend that you use **DirectXMath** instead.

Computes the intersection of a ray and a triangle.

Syntax

```C++
BOOL D3DXIntersectTri(
  _In_  const D3DXVECTOR3 *p0,
  _In_  const D3DXVECTOR3 *p1,
  _In_  const D3DXVECTOR3 *p2,
  _In_  const D3DXVECTOR3 *pRayPos,
  _In_  const D3DXVECTOR3 *pRayDir,
  _Out_       FLOAT       *pU,
  _Out_       FLOAT       *pV,
  _Out_       FLOAT       *pDist
);
```

Parameters

p0 [in]

Type: **const** D3DXVECTOR3*

Pointer to a D3DXVECTOR3 structure, describing the first triangle vertex position.

p1 [in]

Type: **const** D3DXVECTOR3*

Pointer to a D3DXVECTOR3 structure, describing the second triangle vertex position.

p2 [in]

Type: **const** D3DXVECTOR3*

Pointer to a D3DXVECTOR3 structure, describing the third triangle vertex position.

pRayPos [in]

Type: **const** D3DXVECTOR3*

Pointer to a D3DXVECTOR3 structure, specifying the point where the ray begins.

pRayDir [in]

Type: **const** D3DXVECTOR3*

Pointer to a D3DXVECTOR3 structure, specifying the direction of the ray.

pU [out]

Type: FLOAT*

Barycentric hit coordinates, U.

pV [out]

Type: FLOAT*

Barycentric hit coordinates, V.

pDist [out]

Type: FLOAT*

Ray-intersection parameter distance.

Return value

Type: BOOL

Returns **TRUE** if the ray intersects the area of the triangle. Otherwise, returns **FALSE**.

Remarks

Any point in the plane V1V2V3 can be represented by the barycentric coordinate (U,V). The parameter U controls how much V2 gets weighted into the result, and the parameter V controls how much V3 gets weighted into the result. Lastly, the value of [1 - (U + V)] controls how much V1 gets weighted into the result.

Barycentric coordinates are a form of general coordinates. In this context, using barycentric coordinates represents a change in coordinate systems. What holds true for Cartesian coordinates holds true for barycentric coordinates.

Barycentric coordinates define a point inside a triangle in terms of the triangle's vertices. For a more in-depth description of barycentric coordinates, see Mathworld's Barycentric Coordinates Description.

Requirements

Requirement	Value
Header	D3DX10math.h
Library	D3DX10.lib

See also

Mesh Functions

D3DXSphereBoundProbe function (D3DX10math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Determines if a ray intersects the volume of a sphere's bounding box.

Syntax

```C++
BOOL D3DXSphereBoundProbe(
  _In_ const D3DXVECTOR3 *pCenter,
  _In_       FLOAT       Radius,
  _In_ const D3DXVECTOR3 *pRayPosition,
  _In_ const D3DXVECTOR3 *pRayDirection
);
```

Parameters

pCenter [in]

Type: **const** **D3DXVECTOR3***

Pointer to a **D3DXVECTOR3** structure, specifying the center coordinate of the sphere.

Radius [in]

Type: **FLOAT**

Radius of the sphere.

pRayPosition [in]

Type: **const** **D3DXVECTOR3***

Pointer to a **D3DXVECTOR3** structure, specifying the origin coordinate of the ray.

pRayDirection [in]

Type: **const** D3DXVECTOR3*

Pointer to a D3DXVECTOR3 structure, specifying the direction of the ray. This vector should not be (0,0,0) but does not need to be normalized.

Return value

Type: BOOL

Returns **TRUE** if the ray intersects the volume of the sphere's bounding box. Otherwise, returns **FALSE**.

Remarks

D3DXSphereBoundProbe determines if the ray intersects the volume of the sphere's bounding box, not just the surface of the sphere.

Requirements

Requirement	Value
Header	D3DX10math.h
Library	D3DX10.lib

See also

Mesh Functions

Texture Functions in D3DX 10

Article • 01/06/2021 • 2 minutes to read

D3DX supplies the following texture functions.

- D3DX10ComputeNormalMap
- D3DX10CreateAsyncTextureInfoProcessor
- D3DX10CreateAsyncTextureProcessor
- D3DX10CreateShaderResourceViewFromFile
- D3DX10CreateShaderResourceViewFromMemory
- D3DX10CreateShaderResourceViewFromResource
- D3DX10CreateTextureFromFile
- D3DX10CreateTextureFromMemory
- D3DX10CreateTextureFromResource
- D3DX10FilterTexture
- D3DX10GetImageInfoFromFile
- D3DX10GetImageInfoFromMemory
- D3DX10GetImageInfoFromResource
- D3DX10LoadTextureFromTexture
- D3DX10SaveTextureToFile
- D3DX10SaveTextureToMemory

Related topics

D3DX Functions

D3DX10ComputeNormalMap function

Article • 04/27/2021 • 2 minutes to read

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

C++

```
HRESULT D3DX10ComputeNormalMap(
  _In_ ID3D10Texture2D *pSrcTexture,
  _In_ UINT            Flags,
  _In_ UINT            Channel,
  _In_ FLOAT           Amplitude,
  _In_ ID3D10Texture2D *pDestTexture
);
```

Parameters

pSrcTexture [in]

Type: **ID3D10Texture2D***

Pointer to an ID3D10Texture2D 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: **ID3D10Texture2D***

Pointer to an ID3D10Texture2D 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

Requirement	Value
Header	D3DX10Tex.h
Library	D3DX10.lib

See also

Texture Functions in D3DX 10

D3DX10CreateAsyncTextureInfoProcessor function

Article • 04/27/2021 • 2 minutes to read

Create a data processor to be used with a **thread pump**.

Syntax

```
C++

HRESULT D3DX10CreateAsyncTextureInfoProcessor(
  _In_  D3DX10_IMAGE_LOAD_INFO *pLoadInfo,
  _Out_ ID3DX10DataProcessor   **ppDataProcessor
);
```

Parameters

pLoadInfo [in]

Type: **D3DX10_IMAGE_LOAD_INFO***

Optional. Identifies the characteristics of a texture (see **D3DX10_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: **ID3DX10DataProcessor****

Address of a pointer to a buffer that contains the data processor created (see **ID3DX10DataProcessor Interface**).

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Remarks

This API does creates a data-processor interface; **D3DX10CreateAsyncTextureProcessor** creates the data-processor interface and loads the texture.

Requirements

Requirement	Value
Header	D3DX10Tex.h
Library	D3DX10.lib

See also

Texture Functions in D3DX 10

D3DX10CreateAsyncTextureProcessor function

Article • 04/27/2021 • 2 minutes to read

Create a data processor to be used with a thread pump.

Syntax

C++

```
HRESULT D3DX10CreateAsyncTextureProcessor(
  _In_  ID3D10Device           *pDevice,
  _In_  D3DX10_IMAGE_LOAD_INFO *pLoadInfo,
  _Out_ ID3DX10DataProcessor   **ppDataProcessor
);
```

Parameters

pDevice [in]

Type: ID3D10Device*

A pointer to the devive (see ID3D10Device Interface).

pLoadInfo [in]

Type: D3DX10_IMAGE_LOAD_INFO*

Optional. Identifies the characteristics of a texture (see D3DX10_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: ID3DX10DataProcessor**

Address of a pointer to a buffer that contains the data processor created (see ID3DX10DataProcessor Interface).

Return value

Type: HRESULT

The return value is one of the values listed in Direct3D 10 Return Codes.

Remarks

This API does creates a data-processor interface and loads the texture; **D3DX10CreateAsyncTextureInfoProcessor** creates the data-processor interface.

Requirements

Requirement	Value
Header	D3DX10Tex.h
Library	D3DX10.lib

See also

Texture Functions in D3DX 10

D3DX10CreateShaderResourceViewFromFile function

Article • 03/15/2021 • 2 minutes to read

Create a shader-resource view from a file.

Syntax

C++

```
HRESULT D3DX10CreateShaderResourceViewFromFile(
  _In_  ID3D10Device             *pDevice,
  _In_  LPCTSTR                  pSrcFile,
  _In_  D3DX10_IMAGE_LOAD_INFO   *pLoadInfo,
  _In_  ID3DX10ThreadPump        *pPump,
  _Out_ ID3D10ShaderResourceView **ppShaderResourceView,
  _Out_ HRESULT                  *pHResult
);
```

Parameters

pDevice [in]

Type: **ID3D10Device***

A pointer to the device (see **ID3D10Device Interface**) 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.

pLoadInfo [in]

Type: **D3DX10_IMAGE_LOAD_INFO***

Optional. Identifies the characteristics of a texture (see **D3DX10_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: **ID3DX10ThreadPump***

Pointer to a thread-pump interface (see **ID3DX10ThreadPump Interface**). If **NULL** is specified, this function will behave synchronously and will not return until it is finished.

ppShaderResourceView [out]

Type: **ID3D10ShaderResourceView****

Address of a pointer to the shader-resource view (see **ID3D10ShaderResourceView Interface**).

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 10 Return Codes.

Remarks

For a list of supported image formats, see **D3DX10_IMAGE_FILE_FORMAT**.

Requirements

Requirement	Value
Header	D3DX10Tex.h
Library	D3DX10.lib

See also

Texture Functions in D3DX 10

General Purpose Functions

D3DX10CreateShaderResourceViewFromMemory function

Article • 03/15/2021 • 2 minutes to read

Create a shader-resource view from a file in memory.

Syntax

C++

```
HRESULT D3DX10CreateShaderResourceViewFromMemory(
  _In_  ID3D10Device             *pDevice,
  _In_  LPCVOID                  pSrcData,
  _In_  SIZE_T                   SrcDataSize,
  _In_  D3DX10_IMAGE_LOAD_INFO   *pLoadInfo,
  _In_  ID3DX10ThreadPump        *pPump,
  _Out_ ID3D10ShaderResourceView **ppShaderResourceView,
  _Out_ HRESULT                  *pHResult
);
```

Parameters

pDevice [in]

Type: **ID3D10Device***

A pointer to the device (see **ID3D10Device Interface**) 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.

pLoadInfo [in]

Type: **D3DX10_IMAGE_LOAD_INFO***

Optional. Identifies the characteristics of a texture (see **D3DX10_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: **ID3DX10ThreadPump***

A pointer to a thread pump interface (see **ID3DX10ThreadPump Interface**). If **NULL** is specified, this function will behave synchronously and will not return until it is finished.

ppShaderResourceView [out]

Type: **ID3D10ShaderResourceView****

Address of a pointer to the newly created shader resource view. See **ID3D10ShaderResourceView Interface**.

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 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10Tex.h
Library	D3DX10.lib

See also

Texture Functions in D3DX 10

General Purpose Functions

D3DX10CreateShaderResourceViewFromResource function

Article • 03/15/2021 • 2 minutes to read

Create a shader-resource view from a resource.

Syntax

C++

```
HRESULT D3DX10CreateShaderResourceViewFromResource(
  _In_  ID3D10Device             *pDevice,
  _In_  HMODULE                  hSrcModule,
  _In_  LPCTSTR                  pSrcResource,
  _In_  D3DX10_IMAGE_LOAD_INFO   *pLoadInfo,
  _In_  ID3DX10ThreadPump        *pPump,
  _Out_ ID3D10ShaderResourceView **ppShaderResourceView,
  _Out_ HRESULT                  *pHResult
);
```

Parameters

pDevice [in]

Type: **ID3D10Device***

A pointer to the device (see **ID3D10Device Interface**) 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: **D3DX10_IMAGE_LOAD_INFO***

Optional. Identifies the characteristics of a texture (see **D3DX10_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: **ID3DX10ThreadPump***

A pointer to a thread pump interface (see **ID3DX10ThreadPump Interface**). If **NULL** is specified, this function will behave synchronously and will not return until it is finished.

ppShaderResourceView [out]

Type: **ID3D10ShaderResourceView****

Address of a pointer to the shader-resource view (see **ID3D10ShaderResourceView Interface**).

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 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10Tex.h
Library	D3DX10.lib

See also

Texture Functions in D3DX 10

General Purpose Functions

D3DX10CreateTextureFromFile function

Article • 03/15/2021 • 2 minutes to read

Create a texture resource from a file.

Syntax

```
C++

HRESULT D3DX10CreateTextureFromFile(
  _In_  ID3D10Device           *pDevice,
  _In_  LPCTSTR                pSrcFile,
  _In_  D3DX10_IMAGE_LOAD_INFO *pLoadInfo,
  _In_  ID3DX10ThreadPump      *pPump,
  _Out_ ID3D10Resource         **ppTexture,
  _Out_ HRESULT                *pHResult
);
```

Parameters

pDevice [in]

Type: **ID3D10Device***

A pointer to the device (see **ID3D10Device Interface**) 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. See **D3DX10_IMAGE_FILE_FORMAT** enumeration for a list of the supported image file formats.

pLoadInfo [in]

Type: **D3DX10_IMAGE_LOAD_INFO***

Optional. Identifies the characteristics of a texture (see **D3DX10_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: **ID3DX10ThreadPump***

A pointer to a thread pump interface (see **ID3DX10ThreadPump Interface**). If **NULL** is specified, this function will behave synchronously and will not return until it is finished.

ppTexture [out]

Type: **ID3D10Resource****

The address of a pointer to the texture resource (see **ID3D10Resource Interface**).

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 10 Return Codes.

Remarks

For a list of supported image formats see **D3DX10_IMAGE_FILE_FORMAT**.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

Texture Functions in D3DX 10

General Purpose Functions

D3DX10CreateTextureFromMemory function

Article • 03/15/2021 • 2 minutes to read

Create a texture resource from a file residing in system memory.

Syntax

C++

```
HRESULT D3DX10CreateTextureFromMemory(
  _In_  ID3D10Device           *pDevice,
  _In_  LPCVOID                pSrcData,
  _In_  SIZE_T                 SrcDataSize,
  _In_  D3DX10_IMAGE_LOAD_INFO *pLoadInfo,
  _In_  ID3DX10ThreadPump      *pPump,
  _Out_ ID3D10Resource         **ppTexture,
  _Out_ HRESULT                *pHResult
);
```

Parameters

pDevice [in]

Type: **ID3D10Device***

A pointer to the device (see **ID3D10Device Interface**) 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.

pLoadInfo [in]

Type: **D3DX10_IMAGE_LOAD_INFO***

Optional. Identifies the characteristics of a texture (see **D3DX10_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: **ID3DX10ThreadPump***

A pointer to a thread pump interface (see **ID3DX10ThreadPump Interface**). If **NULL** is specified, this function will behave synchronously and will not return until it is finished.

ppTexture [out]

Type: **ID3D10Resource****

Address of a pointer to the created resource. See **ID3D10Resource Interface**.

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 10 Return Codes.

Remarks

For a list of supported image formats see **D3DX10_IMAGE_FILE_FORMAT**.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

Texture Functions in D3DX 10

General Purpose Functions

D3DX10CreateTextureFromResource function

Article • 03/15/2021 • 2 minutes to read

Create a texture from another resource.

Syntax

C++

```
HRESULT D3DX10CreateTextureFromResource(
  _In_  ID3D10Device           *pDevice,
  _In_  HMODULE                hSrcModule,
  _In_  LPCTSTR                pSrcResource,
  _In_  D3DX10_IMAGE_LOAD_INFO *pLoadInfo,
  _In_  ID3DX10ThreadPump      *pPump,
  _Out_ ID3D10Resource         **ppTexture,
  _Out_ HRESULT                *pHResult
);
```

Parameters

pDevice [in]

Type: **ID3D10Device***

A pointer to the device (see **ID3D10Device Interface**) 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.

pLoadInfo [in]

Type: **D3DX10_IMAGE_LOAD_INFO***

Optional. Identifies the characteristics of a texture (see **D3DX10_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: **ID3DX10ThreadPump***

A pointer to a thread pump interface (see **ID3DX10ThreadPump Interface**). If **NULL** is specified, this function will behave synchronously and will not return until it is finished.

ppTexture [out]

Type: **ID3D10Resource****

The address of a pointer to the texture resource (see **ID3D10Resource Interface**).

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 10 Return Codes.

Remarks

For a list of supported image formats see **D3DX10_IMAGE_FILE_FORMAT**.

Requirements

Requirement	Value
Header	D3DX10.h
Library	D3DX10.lib

See also

Texture Functions in D3DX 10

General Purpose Functions

D3DX10FilterTexture function

Article • 03/15/2021 • 2 minutes to read

Generates mipmap chain using a particular texture filter.

Syntax

```
C++

HRESULT D3DX10FilterTexture(
   ID3D10Resource *pTexture,
   UINT           SrcLevel,
   UINT           MipFilter
);
```

Parameters

pTexture

Type: **ID3D10Resource***

The texture object to be filtered. See **ID3D10Resource**.

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 D3DX10_DEFAULT for D3DX10_FILTER_BOX). See **D3DX10_FILTER_FLAG**.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10Tex.h

See also

Texture Functions in D3DX 10

D3DX10GetImageInfoFromFile function

Article • 04/27/2021 • 2 minutes to read

Retrieves information about a given image file.

Syntax

```
C++

HRESULT D3DX10GetImageInfoFromFile(
  _In_  LPCTSTR           pSrcFile,
  _In_  ID3DX10ThreadPump *pPump,
  _In_  D3DX10_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: **ID3DX10ThreadPump***

Optional thread pump that can be used to load the info asynchronously. Can be **NULL**. See **ID3DX10ThreadPump**.

pSrcInfo [in]

Type: **D3DX10_IMAGE_INFO***

Pointer to a D3DX10_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

This function supports both Unicode and ANSI strings.

Requirements

Requirement	Value
Header	D3DX10Tex.h
Library	D3DX10.lib

See also

Texture Functions in D3DX 10

D3DX10GetImageInfoFromMemory function

Article • 03/15/2021 • 2 minutes to read

Get information about an image already loaded into memory.

Syntax

C++

```
HRESULT D3DX10GetImageInfoFromMemory(
  _In_  LPCVOID           pSrcData,
  _In_  SIZE_T            SrcDataSize,
  _In_  ID3DX10ThreadPump *pPump,
  _In_  D3DX10_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: **ID3DX10ThreadPump***

Optional thread pump that can be used to load the info asynchronously. Can be **NULL**. See **ID3DX10ThreadPump**.

pSrcInfo [in]

Type: **D3DX10_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 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10Tex.h
Library	D3DX10.lib

See also

Texture Functions in D3DX 10

D3DX10GetImageInfoFromResource function

Article • 04/27/2021 • 2 minutes to read

Retrieves information about a given image in a resource.

Syntax

C++

```
HRESULT D3DX10GetImageInfoFromResource(
  _In_  HMODULE           hSrcModule,
  _In_  LPCTSTR           pSrcResource,
  _In_  ID3DX10ThreadPump *pPump,
  _In_  D3DX10_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: **ID3DX10ThreadPump***

Optional thread pump that can be used to load the info asynchronously. Can be **NULL**. See **ID3DX10ThreadPump**.

pSrcInfo [in]

Type: **D3DX10_IMAGE_INFO***

Pointer to a D3DX10_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 D3DX10GetImageInfoFromResourceW. Otherwise, the function call resolves to D3DX10GetImageInfoFromResourceA because ANSI strings are being used.

Requirements

Requirement	Value
Header	D3DX10Tex.h
Library	D3DX10.lib

See also

Texture Functions in D3DX 10

D3DX10LoadTextureFromTexture function

Article • 03/15/2021 • 2 minutes to read

Load a texture from a texture.

Syntax

```
C++

HRESULT D3DX10LoadTextureFromTexture(
   ID3D10Resource           *pSrcTexture,
   D3DX10_TEXTURE_LOAD_INFO *pLoadInfo,
   ID3D10Resource           *pDstTexture
);
```

Parameters

pSrcTexture

Type: **ID3D10Resource***

Pointer to the source texture. See **ID3D10Resource**.

pLoadInfo

Type: **D3DX10_TEXTURE_LOAD_INFO***

Pointer to texture loading parameters. See **D3DX10_TEXTURE_LOAD_INFO**.

pDstTexture

Type: **ID3D10Resource***

Pointer to the destination texture. See **ID3D10Resource Interface**.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10Tex.h

See also

Texture Functions in D3DX 10

D3DX10SaveTextureToFile function

Article • 03/15/2021 • 2 minutes to read

Save a texture to a file.

Syntax

```
C++

HRESULT D3DX10SaveTextureToFile(
  _In_ ID3D10Resource           *pSrcTexture,
  _In_ D3DX10_IMAGE_FILE_FORMAT DestFormat,
  _In_ LPCTSTR                  pDestFile
);
```

Parameters

pSrcTexture [in]

Type: **ID3D10Resource***

Pointer to the texture to be saved. See **ID3D10Resource Interface**.

DestFormat [in]

Type: **D3DX10_IMAGE_FILE_FORMAT**

The format the texture will be saved as (see **D3DX10_IMAGE_FILE_FORMAT**). D3DX10_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 10 Return Codes; use the return value to see if the *DestFormat* is supported.

Remarks

D3DX10SaveTextureToFile 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 **D3DX10SaveTextureToFile** 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 prescense of the **DDS_HEADER_DXT10** extended header.

Requirements

Requirement	Value
Header	D3DX10Tex.h
Library	D3DX10.lib

See also

Texture Functions in D3DX 10

General Purpose Functions

D3DX10SaveTextureToMemory function

Article • 03/15/2021 • 2 minutes to read

Save a texture to memory.

Syntax

```
C++

HRESULT D3DX10SaveTextureToMemory(
  _In_  ID3D10Resource           *pSrcTexture,
  _In_  D3DX10_IMAGE_FILE_FORMAT DestFormat,
  _Out_ LPD3D10BLOB              *ppDestBuf,
  _In_  UINT                     Flags
);
```

Parameters

pSrcTexture [in]

Type: **ID3D10Resource***

Pointer to the texture to be saved. See **ID3D10Resource Interface**.

DestFormat [in]

Type: **D3DX10_IMAGE_FILE_FORMAT**

The format the texture will be saved as. See **D3DX10_IMAGE_FILE_FORMAT**.

ppDestBuf [out]

Type: **LPD3D10BLOB***

Address of a pointer to the memory containing the saved texture. See **ID3D10Blob Interface**.

Flags [in]

Type: **UINT**

Optional.

Return value

Type: **HRESULT**

The return value is one of the values listed in Direct3D 10 Return Codes.

Requirements

Requirement	Value
Header	D3DX10Tex.h
Library	D3DX10.lib

See also

Texture Functions in D3DX 10

General Purpose Functions

D3DX Structures (Direct3D 10 Graphics)

Article • 06/19/2021 • 2 minutes to read

This section contains information about the structures included in the D3DX utility library.

Structure	Description
D3DMATRIX	
D3DVECTOR	
D3DX10_ATTRIBUTE_RANGE	Stores an attribute table entry.
D3DX10_ATTRIBUTE_WEIGHTS	Specifies mesh weight attributes.
D3DX10_FONT_DESC	Defines the attributes of a font.
D3DX10_IMAGE_INFO	Returns a description of the original contents of an image file.
D3DX10_IMAGE_LOAD_INFO	
D3DX10_INTERSECT_INFO	Describes a ray-triangle intersection.
D3DX10_SPRITE	Defines position, texture, and color information about a sprite.
D3DX10_TEXTURE_LOAD_INFO	Texture load parameters.
D3DX10_WELD_EPSILONS	Specifies tolerance values for each vertex component when comparing vertices to determine if they are similar enough to be welded together.
D3DX10_SKINNING_CHANNEL	
D3DXCOLOR	Describes color values.
D3DXFLOAT16	Describes a 16-bit floating point vector.
D3DXMATRIX	Describes a matrix.
D3DXMATRIXA16	Defines a 16-byte aligned matrix.
D3DXPLANE	Describes a plane.
D3DXQUATERNION	Describes a quaternion.
D3DXVECTOR2	Describes a vector in 2D space.
D3DXVECTOR2_16F	Same as a D3DXVECTOR2, but it uses 16-bit floating point values for x, y, and z.

Structure	Description
D3DXVECTOR3	Describes a vector in three-dimensional (3D) space.
D3DXVECTOR3_16F	Same as a D3DXVECTOR3, but it uses 16-bit floating point values for x, y, and z.
D3DXVECTOR4	Describes a vector in 4D space.
D3DXVECTOR4_16F	Same as a D3DXVECTOR4, but it uses 16-bit floating point values for x, y, and z.

Related topics

D3DX Reference

D3DMATRIX structure

Article • 03/15/2021 • 2 minutes to read

A 4x4 row-major matrix.

Syntax

```C++
typedef struct D3DMATRIX {
  float m[4][4];
} D3DMATRIX, *LPD3DMATRIX;
```

Members

m[4][4]

Type: **float**

The 4x4 row-major matrix.

Requirements

Requirement	Value
Header	D3DX10Math.h

See also

D3DX Structures

D3DVECTOR structure (D3DX10Math.h)

Article • 03/15/2021 • 2 minutes to read

A 3D vector.

Syntax

```C++
typedef struct D3DVECTOR {
  float x;
  float y;
  float z;
} D3DVECTOR, *LPD3DVECTOR;
```

Members

x

Type: **float**

The x component.

y

Type: **float**

The y component.

z

Type: **float**

The z component.

Requirements

Requirement	Value
Header	D3DX10Math.h

See also

D3DX10_ATTRIBUTE_RANGE structure

Article • 04/27/2021 • 2 minutes to read

Stores an attribute table entry.

Syntax

C++

```
typedef struct D3DX10_ATTRIBUTE_RANGE {
  UINT AttribId;
  UINT FaceStart;
  UINT FaceCount;
  UINT VertexStart;
  UINT VertexCount;
} D3DX10_ATTRIBUTE_RANGE, *LPD3DX10_ATTRIBUTE_RANGE;
```

Members

AttribId

Type: **UINT**

Attribute table identifier.

FaceStart

Type: **UINT**

Starting face.

FaceCount

Type: **UINT**

Face count.

VertexStart

Type: **UINT**

Starting vertex.

VertexCount

Type: **UINT**

Vertex count.

Remarks

An attribute table is used to identify areas of the mesh that need to be drawn with different textures, render states, materials, and so on. In addition, the application can use the attribute table to hide portions of a mesh by not drawing a given attribute identifier (AttribId) when drawing the frame.

The LPD3DX_ATTRIBUTE_RANGE type is defined as a pointer to the D3DX_ATTRIBUTE_RANGE structure.

```
typedef D3DX_ATTRIBUTE_RANGE* LPD3DX_ATTRIBUTE_RANGE;
```

Requirements

Requirement	Value
Header	D3DX10.h

See also

D3DX Structures

D3DX10_ATTRIBUTE_WEIGHTS structure

Article • 04/27/2021 • 2 minutes to read

Specifies mesh weight attributes.

Syntax

C++

```
typedef struct D3DX10_ATTRIBUTE_WEIGHTS {
  FLOAT Position;
  FLOAT Boundary;
  FLOAT Normal;
  FLOAT Diffuse;
  FLOAT Specular;
  FLOAT Texcoord[8];
  FLOAT Tangent;
  FLOAT Binormal;
} D3DX10_ATTRIBUTE_WEIGHTS, *LPD3DX10_ATTRIBUTE_WEIGHTS;
```

Members

Position

Type: **FLOAT**

Position.

Boundary

Type: **FLOAT**

Blend weight.

Normal

Type: **FLOAT**

Normal.

Diffuse

Type: **FLOAT**

Diffuse lighting value.

Specular

Type: **FLOAT**

Specular lighting value.

Texcoord

Type: **FLOAT**

Eight texture coordinates.

Tangent

Type: **FLOAT**

Tangent.

Binormal

Type: **FLOAT**

Binormal.

Remarks

This structure describes how a simplification operation will consider vertex data when calculating relative costs between collapsing edges. For example, if the Normal field is 0.0, the simplification operation will ignore the vertex normal component when calculating the error for the collapse. However, if the Normal field is 1.0, the simplification operation will use the vertex normal component. If the Normal field is 2.0, double the amount of errors; if the Normal field is 4.0, then quadruple the number of errors, and so on.

The LPD3DX_ATTRIBUTE_WEIGHTS type is defined as a pointer to the D3DX_ATTRIBUTE_WEIGHTS structure.

```
typedef D3DX_ATTRIBUTE_WEIGHTS* LPD3DX_ATTRIBUTE_WEIGHTS;
```

Requirements

Requirement	Value

Requirement	Value
Header	D3DX10.h

See also

D3DX Structures

D3DX10_FONT_DESC structure

Article • 03/15/2021 • 2 minutes to read

Defines font attributes.

Syntax

C++

```
typedef struct D3DX10_FONT_DESC {
  INT   Height;
  UINT  Width;
  UINT  Weight;
  UINT  MipLevels;
  BOOL  Italic;
  BYTE  CharSet;
  BYTE  OutputPrecision;
  BYTE  Quality;
  BYTE  PitchAndFamily;
  TCHAR FaceName[LF_FACESIZE];
} D3DX10_FONT_DESC, *LPD3DX10_FONT_DESC;
```

Members

Height

Type: **INT**

Height, in logical units, of the font's character cell or character.

Width

Type: **UINT**

Width, in logical units, of characters in the font.

Weight

Type: **UINT**

Weight of the font in the range from 0 through 1000.

MipLevels

Type: **UINT**

Number of mipmap levels requested. If this value is zero or D3DX_DEFAULT, a complete mipmap chain is created. If the value is 1, the texture space is mapped identically to the screen space.

Italic

Type: **BOOL**

Set to **TRUE** for an Italic font.

CharSet

Type: **BYTE**

Character set.

OutputPrecision

Type: **BYTE**

Output precision. The output precision defines how closely the output must match the requested font height, width, character orientation, escapement, pitch, and font type.

Quality

Type: **BYTE**

Output quality.

PitchAndFamily

Type: **BYTE**

Pitch and family of the font.

FaceName[LF_FACESIZE]

Type: **TCHAR**

A NULL-terminated string that specifies the typeface name of the font. The length of the string must not exceed 32 characters, including the terminating **NULL** character. If FaceName is an empty string, the first font that matches the other specified attributes will be used. If the compiler settings require Unicode, the data type TCHAR resolves to WCHAR; otherwise, the data type resolves to CHAR. See Remarks.

Remarks

The compiler setting also determines the structure type. If Unicode is defined, the D3DX10_FONT_DESC structure type resolves to a D3DX10_FONT_DESCW; otherwise the structure type resolves to a D3DX10_FONT_DESCA.

Possible values of the above members are given in the GDI LOGFONT structure.

Requirements

Requirement	Value
Header	D3DX10.h

See also

D3DX Structures

D3DX10_IMAGE_INFO structure

Article • 04/27/2021 • 2 minutes to read

Returns a description of the original contents of an image file.

Syntax

C++

```
typedef struct D3DX10_IMAGE_INFO {
  UINT                     Width;
  UINT                     Height;
  UINT                     Depth;
  UINT                     ArraySize;
  UINT                     MipLevels;
  UINT                     MiscFlags;
  DXGI_FORMAT              Format;
  D3D10_RESOURCE_DIMENSION ResourceDimension;
  D3DX10_IMAGE_FILE_FORMAT ImageFileFormat;
} D3DX10_IMAGE_INFO, *LPD3DX10_IMAGE_INFO;
```

Members

Width

Type: **UINT**

Width of original image in pixels.

Height

Type: **UINT**

Height of original image in pixels.

Depth

Type: **UINT**

Depth of original image in pixels.

ArraySize

Type: **UINT**

Size of the texture array. *ArraySize* will be 1 for a single image.

MipLevels

Type: **UINT**

Number of mipmap levels in original image.

MiscFlags

Type: **UINT**

Miscellaneous resource properties (see **D3D10_RESOURCE_MISC_FLAG**).

Format

Type: **DXGI_FORMAT**

A value from the **DXGI_FORMAT** enumerated type that most closely describes the data in the original image.

ResourceDimension

Type: **D3D10_RESOURCE_DIMENSION**

Represents the type of the texture stored in the file. See **D3D10_RESOURCE_DIMENSION**.

ImageFileFormat

Type: **D3DX10_IMAGE_FILE_FORMAT**

Represents the format of the image file. See **D3DX10_IMAGE_FILE_FORMAT**.

Requirements

Requirement	Value
Header	D3DX10.h

See also

D3DX Structures

D3DX10_IMAGE_LOAD_INFO structure

Article • 03/15/2021 • 2 minutes to read

Optionally provide information to texture loader APIs to control how textures get loaded. A value of D3DX10_DEFAULT for any of these parameters will cause D3DX to automatically use the value from the source file.

Syntax

C++

```
typedef struct D3DX10_IMAGE_LOAD_INFO {
  UINT              Width;
  UINT              Height;
  UINT              Depth;
  UINT              FirstMipLevel;
  UINT              MipLevels;
  D3D10_USAGE       Usage;
  UINT              BindFlags;
  UINT              CpuAccessFlags;
  UINT              MiscFlags;
  DXGI_FORMAT       Format;
  UINT              Filter;
  UINT              MipFilter;
  D3DX10_IMAGE_INFO *pSrcInfo;
} D3DX10_IMAGE_LOAD_INFO, *LPD3DX10_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 that the texture will have. Using 0 or D3DX10_DEFAULT will cause a full mipmap chain to be created.

Usage

Type: **D3D10_USAGE**

The way the texture resource is intended to be used. See **D3D10_USAGE**.

BindFlags

Type: **UINT**

The pipeline stages that the texture will be allowed to bind to. See **D3D10_BIND_FLAG**.

CpuAccessFlags

Type: **UINT**

The access permissions the cpu will have for the texture resource. See **D3D10_CPU_ACCESS_FLAG**.

MiscFlags

Type: **UINT**

Miscellaneous resource properties (see **D3D10_RESOURCE_MISC_FLAG**).

Format

Type: **DXGI_FORMAT**

The format the texture will be in after it is loaded. See **DXGI_FORMAT**.

Filter

Type: UINT

Filter the texture using the specified filter (only when resampling). See D3DX10_FILTER_FLAG.

MipFilter

Type: UINT

Filter the texture mip levels using the specified filter (only if generating mipmaps). Valid values are D3DX10_FILTER_NONE, D3DX10_FILTER_POINT, D3DX10_FILTER_LINEAR, or D3DX10_FILTER_TRIANGLE. See D3DX10_FILTER_FLAG.

pSrcInfo

Type: D3DX10_IMAGE_INFO*

Information about the original image. See D3DX10_IMAGE_INFO. Can be obtained with D3DX10GetImageInfoFromFile, D3DX10GetImageInfoFromMemory, or D3DX10GetImageInfoFromResource.

Remarks

When initializing the structure, you may set any member to D3DX10_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 D3DX10CreateTextureFromFile and D3DX10CreateShaderResourceViewFromFile.
- Create data processors, such as D3DX10CreateAsyncTextureInfoProcessor or D3DX10CreateAsyncShaderResourceViewProcessor.

Requirements

Requirement	Value
Header	D3DX10Tex.h

See also

D3DX10_INTERSECT_INFO structure

Article • 04/27/2021 • 2 minutes to read

Describes a ray-triangle intersection.

Syntax

C++

```
typedef struct D3DX10_INTERSECT_INFO {
  UINT  FaceIndex;
  FLOAT U;
  FLOAT V;
  FLOAT Dist;
} D3DX10_INTERSECT_INFO, *LPD3DX10_INTERSECT_INFO;
```

Members

FaceIndex

Type: **UINT**

Index of the triangle that hit the ray.

U

Type: **FLOAT**

Barycentric coordinate within the triangle where the ray intersects.

V

Type: **FLOAT**

Barycentric coordinate within the triangle where the ray intersects.

Dist

Type: **FLOAT**

Distance along the ray where the intersection occurred.

Remarks

Barycentric coordinates define a point inside a triangle in terms of the triangle's vertices. For a more in-depth description of barycentric coordinates, see Mathworld's Barycentric Coordinates Description.

Requirements

Requirement	Value
Header	D3DX10.h

See also

D3DX Structures

D3DX10_WELD_EPSILONS structure

Article • 04/27/2021 • 2 minutes to read

Specifies tolerance values for each vertex component when comparing vertices to determine if they are similar enough to be welded together.

Syntax

C++

```
typedef struct D3DX10_WELD_EPSILONS {
  FLOAT Position;
  FLOAT BlendWeights;
  FLOAT Normal;
  FLOAT PSize;
  FLOAT Specular;
  FLOAT Diffuse;
  FLOAT Texcoord[8];
  FLOAT Tangent;
  FLOAT Binormal;
  FLOAT TessFactor;
} D3DX10_WELD_EPSILONS, *LPD3DX10_WELD_EPSILONS;
```

Members

Position

Type: **FLOAT**

Position

BlendWeights

Type: **FLOAT**

Blend weight

Normal

Type: **FLOAT**

Normal

PSize

Type: **FLOAT**

Point size value

Specular

Type: **FLOAT**

Specular lighting value

Diffuse

Type: **FLOAT**

Diffuse lighting value

Texcoord

Type: **FLOAT**

Eight texture coordinates

Tangent

Type: **FLOAT**

Tangent

Binormal

Type: **FLOAT**

Binormal

TessFactor

Type: **FLOAT**

Tessellation factor

Remarks

The LPD3DXWeldEpsilons type is defined as a pointer to the D3DXWeldEpsilons structure.

```
typedef D3DX_WELD_EPSILONS *LPD3DX_WELD_EPSILONS;
```

Requirements

Requirement	Value
Header	D3DX10.h

See also

D3DX Structures

D3DX10_SPRITE structure

Article • 03/15/2021 • 2 minutes to read

Defines position, texture, and color information about a sprite.

Syntax

C++

```
typedef struct D3DX10_SPRITE {
  D3DXMATRIX                matWorld;
  D3DXVECTOR2               TexCoord;
  D3DXVECTOR2               TexSize;
  D3DXCOLOR                 ColorModulate;
  ID3D10ShaderResourceView *pTexture;
  UINT                      TextureIndex;
} D3DX10_SPRITE;
```

Members

matWorld

Type: D3DXMATRIX

The sprite's model-world transformation. This defines the position and orientation of the sprite in world space.

TexCoord

Type: D3DXVECTOR2

Offset from the upper-left corner of the texture indicating where the sprite image should start in the texture. **TexCoord** is in texture coordinates.

TexSize

Type: D3DXVECTOR2

A vector containing the width and height of the sprite in texture coordinates.

ColorModulate

Type: D3DXCOLOR

A color that will be multiplied with the pixel color before rendering.

pTexture

Type: **ID3D10ShaderResourceView***

Pointer to a shader-resource view representing the sprite's texture. See **ID3D10ShaderResourceView Interface**.

TextureIndex

Type: **UINT**

The index of the texture. If pTexture does not represent a texture array, then this should be 0.

Requirements

Requirement	Value
Header	D3DX10.h

See also

D3DX Structures

D3DX10_SKINNING_CHANNEL structure

Article • 03/15/2021 • 2 minutes to read

The member of the vertex decl to do the software skinning on. This is used with the ID3DX10SkinInfo::DoSoftwareSkinning API.

Syntax

```
C++

typedef struct D3DX10_SKINNING_CHANNEL {
  UINT SrcOffset;
  UINT DestOffset;
  BOOL IsNormal;
} D3DX10_SKINNING_CHANNEL, *LPD3DX10_SKINNING_CHANNEL;
```

Members

SrcOffset

Type: UINT

Offset from the beginning of each source vertex.

DestOffset

Type: UINT

Offset from the beginning of each destination vertex.

IsNormal

Type: BOOL

Determines which array of matrices to use in the ID3DX10SkinInfo::DoSoftwareSkinning API. If this is true, the *pInverseTransposeBoneMatrices* will be used, otherwise *pBoneMatrices* will be used.

Requirements

Requirement	Value

Requirement	Value
Header	D3DX10.h

See also

D3DX Structures

D3DX10_TEXTURE_LOAD_INFO structure

Article • 03/15/2021 • 2 minutes to read

Describes parameters used to load a texture from another texture.

Syntax

C++

```
typedef struct _D3DX10_TEXTURE_LOAD_INFO {
  D3D10_BOX *pSrcBox;
  D3D10_BOX *pDstBox;
  UINT      SrcFirstMip;
  UINT      DstFirstMip;
  UINT      NumMips;
  UINT      SrcFirstElement;
  UINT      DstFirstElement;
  UINT      NumElements;
  UINT      Filter;
  UINT      MipFilter;
} D3DX10_TEXTURE_LOAD_INFO;
```

Members

pSrcBox

Type: D3D10_BOX*

Source texture box (see D3D10_BOX).

pDstBox

Type: D3D10_BOX*

Destination texture box (see D3D10_BOX).

SrcFirstMip

Type: UINT

Source texture mipmap level, see D3D10CalcSubresource for more detail.

DstFirstMip

Type: UINT

Destination texture mipmap level, see D3D10CalcSubresource 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 D3DX10_FILTER_FLAG).

MipFilter

Type: UINT

Filtering options when generating mip levels (see D3DX10_FILTER_FLAG).

Remarks

This structure is used in a call to D3DX10LoadTextureFromTexture.

Requirements

Requirement	Value
Header	D3DX10Tex.h

See also

D3DX Structures

D3DXCOLOR structure (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Describes color values.

Syntax

```C++
typedef struct D3DXCOLOR {
  FLOAT r;
  FLOAT g;
  FLOAT b;
  FLOAT a;
} D3DXCOLOR, *LPD3DXCOLOR;
```

Members

r

Type: **FLOAT**

Red component of the color.

g

Type: **FLOAT**

Green component of the color.

b

Type: **FLOAT**

Blue component of the color.

a

Type: **FLOAT**

Alpha component of the color.

Requirements

Requirement	Value
Header	D3DX10Math.h

See also

D3DX Structures

D3DXFLOAT16 structure (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Describes a 16-bit floating point value.

Syntax

```C++
typedef struct D3DXFLOAT16 {
  WORD Value;
} D3DXFLOAT16, *LPD3DXFLOAT16;
```

Members

Value

Type: **WORD**

The 16-bit data.

Requirements

Requirement	Value
Header	D3DX10Math.h

See also

D3DX Structures

D3DXMATRIX structure (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX utility library is deprecated. We recommend that you use **DirectXMath** instead.

A 4x4 matrix that contains methods and operator overloads.

Syntax

```
C++

typedef struct D3DXMATRIX {
  FLOAT _ij;
} D3DXMATRIX, *LPD3DXMATRIX;
```

Members

_ij

Type: **FLOAT**

The (i, j) component of the matrix, where i is the row number and j is the column number. For example, _34 means the same as [a_{34}], the component in the third row and fourth column.

Remarks

C programmers cannot use the D3DXMATRIX structure, they must use the D3DMATRIX structure. C++ programmers can take advantage of overloaded constructors and assignment, unary, and binary (including equality) operators.

In D3DX, the _34 element of a projection matrix cannot be a negative number. If your application needs to use a negative value in this location, it should scale the entire projection matrix by -1 instead.

D3DXMATRIX Extensions

D3DXMATRIX has the following C++ extensions.

```
#ifdef __cplusplus
typedef struct D3DXMATRIX : public D3DMATRIX
{
public:
    D3DXMATRIX() {};
    D3DXMATRIX( CONST FLOAT * );
    D3DXMATRIX( CONST D3DMATRIX& );
    D3DXMATRIX( CONST D3DXFLOAT16 * );
    D3DXMATRIX( FLOAT _11, FLOAT _12, FLOAT _13, FLOAT _14,
                FLOAT _21, FLOAT _22, FLOAT _23, FLOAT _24,
                FLOAT _31, FLOAT _32, FLOAT _33, FLOAT _34,
                FLOAT _41, FLOAT _42, FLOAT _43, FLOAT _44 );


    // access grants
    FLOAT& operator () ( UINT Row, UINT Col );
    FLOAT  operator () ( UINT Row, UINT Col ) const;

    // casting operators
    operator FLOAT* ();
    operator CONST FLOAT* () const;

    // assignment operators
    D3DXMATRIX& operator *= ( CONST D3DXMATRIX& );
    D3DXMATRIX& operator += ( CONST D3DXMATRIX& );
    D3DXMATRIX& operator -= ( CONST D3DXMATRIX& );
    D3DXMATRIX& operator *= ( FLOAT );
    D3DXMATRIX& operator /= ( FLOAT );

    // unary operators
    D3DXMATRIX operator + () const;
    D3DXMATRIX operator - () const;

    // binary operators
    D3DXMATRIX operator * ( CONST D3DXMATRIX& ) const;
    D3DXMATRIX operator + ( CONST D3DXMATRIX& ) const;
    D3DXMATRIX operator - ( CONST D3DXMATRIX& ) const;
    D3DXMATRIX operator * ( FLOAT ) const;
    D3DXMATRIX operator / ( FLOAT ) const;

    friend D3DXMATRIX operator * ( FLOAT, CONST D3DXMATRIX& );

    BOOL operator == ( CONST D3DXMATRIX& ) const;
    BOOL operator != ( CONST D3DXMATRIX& ) const;

} D3DXMATRIX, *LPD3DXMATRIX;

#else //!__cplusplus
typedef struct _D3DMATRIX D3DXMATRIX, *LPD3DXMATRIX;
```

```
#endif //!__cplusplus
```

Requirements

Requirement	Value
Header	D3DX10Math.h

See also

D3DX Structures

D3DXMATRIXA16 structure (D3DX10Math.h)

Article • 02/16/2023 • 2 minutes to read

> **Note**
>
> The D3DX utility library is deprecated. We recommend that you use **DirectXMath** instead.

A 4x4, 16-byte-aligned matrix that contains methods and operator overloads.

Syntax

```
C++

typedef struct D3DXMATRIXA16 {
  FLOAT _ij;
} D3DXMATRIXA16, *LPD3DXMATRIXA16;
```

Members

_ij

Type: **FLOAT**

The (i, j) component of the matrix, where i is the row number and j is the column number. For example, _34 means the same as [a_{34}], the component in the third row and fourth column.

Remarks

A 16-byte aligned matrix, when used by D3DX math functions, has been optimized for improved performance on Intel Pentium 4 processors. Matrices are aligned independent of where they are created: on the program stack, in the heap, or in global scope. Alignment is accomplished using __declspec(align(16)), which works with Visual C++ .NET and with Visual C++ 6.0 only when the processor pack is installed. Unfortunately, there is no way to detect the processor pack, so byte alignment is turned on by default only with Visual C++ .NET.

Vectors and quaternions are not byte aligned in D3DX. When using vectors and quaternions with D3DX math functions, use _declspec(align(16)) to generate byte aligned vectors and quaternions, because they will perform significantly better. The definition of _declspec is shown here.

```
#define D3DX_ALIGN16 __declspec(align(16))
```

Other compilers interpret **D3DXMATRIXA16** as D3DXMATRIX. Using this structure on a compiler that does not actually align the matrix can be problematic because it will not expose bugs that ignore alignment. For example, if a **D3DXMATRIXA16** object is inside a structure or class, a memcpy might be done with tight packing (ignoring 16-byte boundaries). This would cause build breaks if the compiler were to sometime add matrix aligning.

D3DXMATRIXA16 extensions

D3DXMATRIXA16 has the following C++ extensions.

```
typedef struct _D3DXMATRIXA16 : public D3DXMATRIX
{
    _D3DXMATRIXA16();
    _D3DXMATRIXA16( CONST FLOAT * f);
    _D3DXMATRIXA16( CONST D3DMATRIX& m);
    _D3DXMATRIXA16( FLOAT _11, FLOAT _12, FLOAT _13, FLOAT _14,
                    FLOAT _21, FLOAT _22, FLOAT _23, FLOAT _24,
                    FLOAT _31, FLOAT _32, FLOAT _33, FLOAT _34,
                    FLOAT _41, FLOAT _42, FLOAT _43, FLOAT _44 );
    void* operator new(size_t s);
    void* operator new[](size_t s);

    // The two operators below are not virtual operators. If you cast
    //   to D3DXMATRIX, do not delete using them
    void operator delete(void* p);
    void operator delete[](void* p);

    struct _D3DXMATRIXA16& operator=(CONST D3DXMATRIX& rhs);
} _D3DXMATRIXA16;

typedef D3DX_ALIGN16 _D3DXMATRIXA16 D3DXMATRIXA16, *LPD3DXMATRIXA16;
```

Requirements

Requirement	Value
Header	D3DX10Math.h

See also

- D3DX Structures

D3DXPLANE structure (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Describes a plane.

Syntax

```C++
typedef struct D3DXPLANE {
  FLOAT a;
  FLOAT b;
  FLOAT c;
  FLOAT d;
} D3DXPLANE, *LPD3DXPLANE;
```

Members

a

Type: **FLOAT**

The a coefficient of the clipping plane in the general plane equation. See Remarks.

b

Type: **FLOAT**

The b coefficient of the clipping plane in the general plane equation. See Remarks.

c

Type: **FLOAT**

The c coefficient of the clipping plane in the general plane equation. See Remarks.

d

Type: **FLOAT**

The d coefficient of the clipping plane in the general plane equation. See Remarks.

Remarks

The members of the **D3DXPLANE** structure take the form of the general plane equation. They fit into the general plane equation so that ax + by + cz + dw = 0.

Requirements

Requirement	Value
Header	D3DX10Math.h

See also

D3DX Structures

D3DXQUATERNION structure (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Describes a quaternion.

Syntax

C++

```
typedef struct D3DXQUATERNION {
  FLOAT x;
  FLOAT y;
  FLOAT z;
  FLOAT w;
} D3DXQUATERNION, *LPD3DXQUATERNION;
```

Members

x

Type: **FLOAT**

The x-component.

y

Type: **FLOAT**

The y-component.

z

Type: **FLOAT**

The z-component.

w

Type: **FLOAT**

The w-component.

Remarks

Quaternions add a fourth element to the [x, y, z] values that define a vector, resulting in arbitrary 4D vectors. However, the following illustrates how each element of a unit quaternion relates to an axis-angle rotation (where q represents a unit quaternion (x, y, z, w), axis is normalized, and theta is the desired CCW rotation about the axis):

```
q.x = sin(theta/2) * axis.x
q.y = sin(theta/2) * axis.y
q.z = sin(theta/2) * axis.z
q.w = cos(theta/2)
```

Requirements

Requirement	Value
Header	D3DX10Math.h

See also

D3DX Structures

D3DXVECTOR2 structure (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Describes a two-component vector including operator overloads and type casts.

Syntax

```C++
typedef struct D3DXVECTOR2 {
  FLOAT x;
  FLOAT y;
} D3DXVECTOR2, *LPD3DXVECTOR2;
```

Members

x

Type: **FLOAT**

The x-component.

y

Type: **FLOAT**

The y-component.

Remarks

D3DXVECTOR2 has the following C++ extensions.

D3DXVECTOR2 Extensions

```
typedef struct D3DXVECTOR2
{
#ifdef __cplusplus
public:
    D3DXVECTOR2() {};
    D3DXVECTOR2( CONST FLOAT * );
    D3DXVECTOR2( CONST D3DXFLOAT16 * );
    D3DXVECTOR2( FLOAT x, FLOAT y );

    // casting
    operator FLOAT* ();
    operator CONST FLOAT* () const;

    // assignment operators
    D3DXVECTOR2& operator += ( CONST D3DXVECTOR2& );
    D3DXVECTOR2& operator -= ( CONST D3DXVECTOR2& );
    D3DXVECTOR2& operator *= ( FLOAT );
    D3DXVECTOR2& operator /= ( FLOAT );

    // unary operators
    D3DXVECTOR2 operator + () const;
    D3DXVECTOR2 operator - () const;

    // binary operators
    D3DXVECTOR2 operator + ( CONST D3DXVECTOR2& ) const;
    D3DXVECTOR2 operator - ( CONST D3DXVECTOR2& ) const;
    D3DXVECTOR2 operator * ( FLOAT ) const;
    D3DXVECTOR2 operator / ( FLOAT ) const;

    friend D3DXVECTOR2 operator * ( FLOAT, CONST D3DXVECTOR2& );

    BOOL operator == ( CONST D3DXVECTOR2& ) const;
    BOOL operator != ( CONST D3DXVECTOR2& ) const;


public:
#endif //__cplusplus
    FLOAT x, y;
} D3DXVECTOR2, *LPD3DXVECTOR2;
```

Requirements

Requirement	Value
Header	D3DX10Math.h

See also

D3DX Structures

D3DXVECTOR2_16F structure

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Describes a two-component vector including operator overloads and type casts. Same as a **D3DXVECTOR2**, but it uses 16-bit floating point values for x, y, and z.

Syntax

C++

```
typedef struct D3DXVECTOR2_16F {
  FLOAT x;
  FLOAT y;
} D3DXVECTOR2_16F, *LPD3DXVECTOR2_16F;
```

Members

x

Type: **FLOAT**

The x-component.

y

Type: **FLOAT**

The y-component.

Remarks

D3DXVECTOR2_16F has the following C++ extensions.

D3DXVECTOR2_16F Extensions

```
typedef struct D3DXVECTOR2_16F
{
#ifdef __cplusplus
public:
    D3DXVECTOR2_16F() {};
    D3DXVECTOR2_16F( CONST FLOAT * );
    D3DXVECTOR2_16F( CONST D3DXFLOAT16 * );
    D3DXVECTOR2_16F( CONST D3DXFLOAT16 &x, CONST D3DXFLOAT16 &y );

    // casting
    operator D3DXFLOAT16* ();
    operator CONST D3DXFLOAT16* () const;

    // binary operators
    BOOL operator == ( CONST D3DXVECTOR2_16F& ) const;
    BOOL operator != ( CONST D3DXVECTOR2_16F& ) const;

public:
#endif //__cplusplus
    D3DXFLOAT16 x, y;

} D3DXVECTOR2_16F, *LPD3DXVECTOR2_16F;
```

Requirements

Requirement	Value
Header	D3DX10Math.h

See also

D3DX Structures

D3DXVECTOR3 structure (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Describes a three-component vector including operator overloads and type casts.

Syntax

C++

```
typedef struct D3DXVECTOR3 {
  FLOAT x;
  FLOAT y;
  FLOAT z;
} D3DXVECTOR3, *LPD3DXVECTOR3;
```

Members

x

Type: **FLOAT**

The x-component.

y

Type: **FLOAT**

The y-component.

z

Type: **FLOAT**

The z-component.

Remarks

D3DXVECTOR3 has the following C++ extensions.

D3DXVECTOR3 Extensions

```
#ifdef __cplusplus
typedef struct D3DXVECTOR3 : public D3DVECTOR
{
public:
    D3DXVECTOR3() {};
    D3DXVECTOR3( CONST FLOAT * );
    D3DXVECTOR3( CONST D3DVECTOR& );
    D3DXVECTOR3( CONST D3DXFLOAT16 * );
    D3DXVECTOR3( FLOAT x, FLOAT y, FLOAT z );

    // casting
    operator FLOAT* ();
    operator CONST FLOAT* () const;

    // assignment operators
    D3DXVECTOR3& operator += ( CONST D3DXVECTOR3& );
    D3DXVECTOR3& operator -= ( CONST D3DXVECTOR3& );
    D3DXVECTOR3& operator *= ( FLOAT );
    D3DXVECTOR3& operator /= ( FLOAT );

    // unary operators
    D3DXVECTOR3 operator + () const;
    D3DXVECTOR3 operator - () const;

    // binary operators
    D3DXVECTOR3 operator + ( CONST D3DXVECTOR3& ) const;
    D3DXVECTOR3 operator - ( CONST D3DXVECTOR3& ) const;
    D3DXVECTOR3 operator * ( FLOAT ) const;
    D3DXVECTOR3 operator / ( FLOAT ) const;

    friend D3DXVECTOR3 operator * ( FLOAT, CONST struct D3DXVECTOR3& );

    BOOL operator == ( CONST D3DXVECTOR3& ) const;
    BOOL operator != ( CONST D3DXVECTOR3& ) const;

} D3DXVECTOR3, *LPD3DXVECTOR3;

#else //!__cplusplus
typedef struct _D3DVECTOR D3DXVECTOR3, *LPD3DXVECTOR3;
#endif //!__cplusplus
```

Requirements

Requirement	Value
Header	D3DX10Math.h

See also

D3DX Structures

D3DXVECTOR3_16F structure

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Same as a **D3DXVECTOR3**, but it uses 16-bit floating point values for x, y, and z.

Syntax

C++

```
typedef struct D3DXVECTOR3_16F {
  FLOAT x;
  FLOAT y;
  FLOAT z;
} D3DXVECTOR3_16F, *LPD3DXVECTOR3_16F;
```

Members

x

Type: **FLOAT**

The x-component.

y

Type: **FLOAT**

The y-component.

z

Type: **FLOAT**

The z-component.

Remarks

D3DXVECTOR3_16F has the following C++ extensions.

D3DXVECTOR3_16F Extensions

```
typedef struct D3DXVECTOR3_16F
{
#ifdef __cplusplus
public:
    D3DXVECTOR3_16F() {};
    D3DXVECTOR3_16F( CONST FLOAT * );
    D3DXVECTOR3_16F( CONST D3DVECTOR& );
    D3DXVECTOR3_16F( CONST D3DXFLOAT16 * );
    D3DXVECTOR3_16F( CONST D3DXFLOAT16 &x, CONST D3DXFLOAT16 &y, CONST
D3DXFLOAT16 &z );

    // casting
    operator D3DXFLOAT16* ();
    operator CONST D3DXFLOAT16* () const;

    // binary operators
    BOOL operator == ( CONST D3DXVECTOR3_16F& ) const;
    BOOL operator != ( CONST D3DXVECTOR3_16F& ) const;

public:
#endif //__cplusplus
    D3DXFLOAT16 x, y, z;

} D3DXVECTOR3_16F, *LPD3DXVECTOR3_16F;
```

Requirements

Requirement	Value
Header	D3DX10Math.h

See also

D3DX Structures

D3DXVECTOR4 structure (D3DX10Math.h)

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Describes a four-component vector including operator overloads and type casts.

Syntax

C++

```
typedef struct D3DXVECTOR4 {
  FLOAT x;
  FLOAT y;
  FLOAT z;
  FLOAT w;
} D3DXVECTOR4, *LPD3DXVECTOR4;
```

Members

x

Type: **FLOAT**

The x-component.

y

Type: **FLOAT**

The y-component.

z

Type: **FLOAT**

The z-component.

w

Type: **FLOAT**

The w-component.

Remarks

D3DXVECTOR4 has the following C++ extensions.

D3DXVECTOR4 Extensions

```
typedef struct D3DXVECTOR4
{
#ifdef __cplusplus
public:
    D3DXVECTOR4() {};
    D3DXVECTOR4( CONST FLOAT* );
    D3DXVECTOR4( CONST D3DXFLOAT16 * );
    D3DXVECTOR4( CONST D3DVECTOR& xyz, FLOAT w );
    D3DXVECTOR4( FLOAT x, FLOAT y, FLOAT z, FLOAT w );

    // casting
    operator FLOAT* ();
    operator CONST FLOAT* () const;

    // assignment operators
    D3DXVECTOR4& operator += ( CONST D3DXVECTOR4& );
    D3DXVECTOR4& operator -= ( CONST D3DXVECTOR4& );
    D3DXVECTOR4& operator *= ( FLOAT );
    D3DXVECTOR4& operator /= ( FLOAT );

    // unary operators
    D3DXVECTOR4 operator + () const;
    D3DXVECTOR4 operator - () const;

    // binary operators
    D3DXVECTOR4 operator + ( CONST D3DXVECTOR4& ) const;
    D3DXVECTOR4 operator - ( CONST D3DXVECTOR4& ) const;
    D3DXVECTOR4 operator * ( FLOAT ) const;
    D3DXVECTOR4 operator / ( FLOAT ) const;

    friend D3DXVECTOR4 operator * ( FLOAT, CONST D3DXVECTOR4& );

    BOOL operator == ( CONST D3DXVECTOR4& ) const;
    BOOL operator != ( CONST D3DXVECTOR4& ) const;

public:
```

```
#endif //__cplusplus
    FLOAT x, y, z, w;
} D3DXVECTOR4, *LPD3DXVECTOR4;
```

Requirements

Requirement	Value
Header	D3DX10Math.h

See also

D3DX Structures

D3DXVECTOR4_16F structure

Article • 04/28/2022 • 2 minutes to read

> **Note**
>
> The D3DX10 utility library is deprecated. We recommend that you use **DirectXMath** instead.

Same as a **D3DXVECTOR4**, but it uses 16-bit floating point values for x, y, and z.

Syntax

C++

```
typedef struct D3DXVECTOR4_16F {
  FLOAT x;
  FLOAT y;
  FLOAT z;
  FLOAT w;
} D3DXVECTOR4_16F, *LPD3DXVECTOR4_16F;
```

Members

x

Type: **FLOAT**

The x-component.

y

Type: **FLOAT**

The y-component.

z

Type: **FLOAT**

The z-component.

w

Type: **FLOAT**

The w-component.

Remarks

D3DXVECTOR4_16F has the following C++ extensions.

D3DXVECTOR4_16F Extensions

```
typedef struct D3DXVECTOR4_16F
{
#ifdef __cplusplus
public:
    D3DXVECTOR4_16F() {};
    D3DXVECTOR4_16F( CONST FLOAT * );
    D3DXVECTOR4_16F( CONST D3DXFLOAT16 * );
    D3DXVECTOR4_16F( CONST D3DXVECTOR3_16F& xyz, CONST D3DXFLOAT16& w );
    D3DXVECTOR4_16F( CONST D3DXFLOAT16& x, CONST D3DXFLOAT16& y, CONST
D3DXFLOAT16& z, CONST D3DXFLOAT16& w );

    // casting
    operator D3DXFLOAT16* ();
    operator CONST D3DXFLOAT16* () const;

    // binary operators
    BOOL operator == ( CONST D3DXVECTOR4_16F& ) const;
    BOOL operator != ( CONST D3DXVECTOR4_16F& ) const;

public:
#endif //__cplusplus
    D3DXFLOAT16 x, y, z, w;

} D3DXVECTOR4_16F, *LPD3DXVECTOR4_16F;
```

Requirements

Requirement	Value
Header	D3DX10Math.h

See also

D3DX Enumerations (Direct3D 10 Graphics)

Article • 01/06/2021 • 2 minutes to read

This section contains information about the following enumerated types and flags used with D3DX.:

Enumeration	Description
D3DX10_CHANNEL_FLAG	These flags are used by functions which operate on one or more channels in a texture.
D3DX_CPU_OPTIMIZATION	Specifies the instruction set D3DX is currently optimized for.
D3DX10_ERR	Errors are represented by negative values and cannot be combined.
D3DX10_FILTER_FLAG	Texture filtering flags.
D3DX10_IMAGE_FILE_FORMAT	Describes the supported image file formats.
D3DX10_MESH	Flags used to specify creation options for a mesh.
D3DX10_MESH_DISCARD_FLAGS	Specifies which pieces of mesh data to discard from the device. Used with ID3DX10Mesh::Discard.
D3DX10_MESHOPT	Specifies the type of mesh optimization to be performed.
D3DX10_NORMALMAP_FLAG	These flags are used to control how D3DX10ComputeNormalMap generates normal maps. Any number of these flags may be OR'd together in any combination.
D3DX10_SAVE_TEXTURE_FLAG	Texture save options.

Related topics

D3DX Reference

D3DX10_CHANNEL_FLAG enumeration

Article • 03/15/2021 • 2 minutes to read

These flags are used by functions which operate on one or more channels in a texture.

Syntax

C++

```
typedef enum D3DX10_CHANNEL_FLAG {
  D3DX10_CHANNEL_RED        = (1 << 0),
  D3DX10_CHANNEL_BLUE       = (1 << 1),
  D3DX10_CHANNEL_GREEN      = (1 << 2),
  D3DX10_CHANNEL_ALPHA      = (1 << 3),
  D3DX10_CHANNEL_LUMINANCE  = (1 << 4)
} D3DX10_CHANNEL_FLAG, *LPD3DX10_CHANNEL_FLAG;
```

Constants

D3DX10_CHANNEL_RED

Indicates the red channel should be used.

D3DX10_CHANNEL_BLUE

Indicates the blue channel should be used.

D3DX10_CHANNEL_GREEN

Indicates the green channel should be used.

D3DX10_CHANNEL_ALPHA

Indicates the alpha channel should be used.

D3DX10_CHANNEL_LUMINANCE

Indicates the luminaces of the red, green, and blue channels should be used.

Requirements

Requirement	Value

Requirement	Value
Header	D3DX10Tex.h

See also

D3DX Enumerations

D3DX_CPU_OPTIMIZATION enumeration

Article • 03/15/2021 • 2 minutes to read

Specifies the instruction set D3DX is currently optimized for.

Syntax

```C++
typedef enum _D3DX_CPU_OPTIMIZATION {
  D3DX_NOT_OPTIMIZED    = 0,
  D3DX_3DNOW_OPTIMIZED  = 1,
  D3DX_SSE2_OPTIMIZED   = 2,
  D3DX_SSE_OPTIMIZED    = 3
} D3DX_CPU_OPTIMIZATION;
```

Constants

D3DX_NOT_OPTIMIZED

Do not optimize.

D3DX_3DNOW_OPTIMIZED

Optimize for a 3DNow! instruction set.

D3DX_SSE2_OPTIMIZED

Optimize for an SSE2 instruction set.

D3DX_SSE_OPTIMIZED

Optimize for an SSE instruction set.

Requirements

Requirement	Value
Header	D3DX10Math.h

See also

D3DX10_ERR enumeration

Article • 04/27/2021 • 2 minutes to read

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

C++

```
typedef enum D3DX10_ERR {
  D3DX10_ERR_CANNOT_MODIFY_INDEX_BUFFER  = MAKE_DDHRESULT(2900),
  D3DX10_ERR_INVALID_MESH                = MAKE_DDHRESULT(2901),
  D3DX10_ERR_CANNOT_ATTR_SORT            = MAKE_DDHRESULT(2902),
  D3DX10_ERR_SKINNING_NOT_SUPPORTED      = MAKE_DDHRESULT(2903),
  D3DX10_ERR_TOO_MANY_INFLUENCES         = MAKE_DDHRESULT(2904),
  D3DX10_ERR_INVALID_DATA                = MAKE_DDHRESULT(2905),
  D3DX10_ERR_LOADED_MESH_HAS_NO_DATA     = MAKE_DDHRESULT(2906),
  D3DX10_ERR_DUPLICATE_NAMED_FRAGMENT    = MAKE_DDHRESULT(2907),
  D3DX10_ERR_CANNOT_REMOVE_LAST_ITEM     = MAKE_DDHRESULT(2908)
} D3DX10_ERR, *LPD3DX10_ERR;
```

Constants

D3DX10_ERR_CANNOT_MODIFY_INDEX_BUFFER

The index buffer cannot be modified.

D3DX10_ERR_INVALID_MESH

The mesh is invalid.

D3DX10_ERR_CANNOT_ATTR_SORT

Attribute sort (D3DXMESHOPT_ATTRSORT) is not supported as an optimization technique.

D3DX10_ERR_SKINNING_NOT_SUPPORTED

Skinning is not supported.

D3DX10_ERR_TOO_MANY_INFLUENCES

Too many influences specified.

D3DX10_ERR_INVALID_DATA

The data is invalid.

D3DX10_ERR_LOADED_MESH_HAS_NO_DATA

The mesh has no data.

D3DX10_ERR_DUPLICATE_NAMED_FRAGMENT

A fragment with that name already exists.

D3DX10_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

Requirement	Value
Header	D3DX10.h

See also

D3DX Enumerations

D3DX10_FILTER_FLAG enumeration

Article • 03/15/2021 • 2 minutes to read

Texture filtering flags.

Syntax

C++

```
typedef enum D3DX10_FILTER_FLAG {
  D3DX10_FILTER_NONE              = (1 << 0),
  D3DX10_FILTER_POINT             = (2 << 0),
  D3DX10_FILTER_LINEAR            = (3 << 0),
  D3DX10_FILTER_TRIANGLE          = (4 << 0),
  D3DX10_FILTER_BOX               = (5 << 0),
  D3DX10_FILTER_MIRROR_U          = (1 << 16),
  D3DX10_FILTER_MIRROR_V          = (2 << 16),
  D3DX10_FILTER_MIRROR_W          = (4 << 16),
  D3DX10_FILTER_MIRROR            = (7 << 16),
  D3DX10_FILTER_DITHER            = (1 << 19),
  D3DX10_FILTER_DITHER_DIFFUSION  = (2 << 19),
  D3DX10_FILTER_SRGB_IN           = (1 << 21),
  D3DX10_FILTER_SRGB_OUT          = (2 << 21),
  D3DX10_FILTER_SRGB              = (3 << 21)
} D3DX10_FILTER_FLAG, *LPD3DX10_FILTER_FLAG;
```

Constants

D3DX10_FILTER_NONE

No scaling or filtering will take place. Pixels outside the bounds of the source image are assumed to be transparent black.

D3DX10_FILTER_POINT

Each destination pixel is computed by sampling the nearest pixel from the source image.

D3DX10_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.

D3DX10_FILTER_TRIANGLE

Every pixel in the source image contributes equally to the destination image. This is the slowest of the filters.

D3DX10_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.

D3DX10_FILTER_MIRROR_U

Pixels off the edge of the texture on the u-axis should be mirrored, not wrapped.

D3DX10_FILTER_MIRROR_V

Pixels off the edge of the texture on the v-axis should be mirrored, not wrapped.

D3DX10_FILTER_MIRROR_W

Pixels off the edge of the texture on the w-axis should be mirrored, not wrapped.

D3DX10_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.

D3DX10_FILTER_DITHER

The resulting image must be dithered using a 4x4 ordered dither algorithm. This happens when converting from one format to another.

D3DX10_FILTER_DITHER_DIFFUSION

Do diffuse dithering on the image when changing from one format to another.

D3DX10_FILTER_SRGB_IN

Input data is in standard RGB (sRGB) color space. See remarks.

D3DX10_FILTER_SRGB_OUT

Output data is in standard RGB (sRGB) color space. See remarks.

D3DX10_FILTER_SRGB

Same as specifying D3DX_FILTER_SRGB_IN | D3DX_FILTER_SRGB_OUT. See remarks.

Remarks

D3DX10 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

Requirement	Value
Header	D3DX10Tex.h

See also

D3DX Enumerations

D3DX10_IMAGE_FILE_FORMAT enumeration

Article • 03/15/2021 • 2 minutes to read

Image file formats supported by D3DXCreatexxx and D3DX10Savexxx functions.

Syntax

C++

```
typedef enum D3DX10_IMAGE_FILE_FORMAT {
  D3DX10_IFF_BMP          = 0,
  D3DX10_IFF_JPG          = 1,
  D3DX10_IFF_PNG          = 3,
  D3DX10_IFF_DDS          = 4,
  D3DX10_IFF_TIFF         = 10,
  D3DX10_IFF_GIF          = 11,
  D3DX10_IFF_WMP          = 12,
  D3DX10_IFF_FORCE_DWORD  = 0x7fffffff
} D3DX10_IMAGE_FILE_FORMAT, *LPD3DX10_IMAGE_FILE_FORMAT;
```

Constants

D3DX10_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.

D3DX10_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.

D3DX10_IFF_PNG

Portable Network Graphics (PNG) file format. A non-proprietary bitmap format using lossless compression. The file extension for this format is .png.

D3DX10_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.

D3DX10_IFF_TIFF

Tagged Image File Format (TIFF). The file extensions for this format are .tif and .tiff.

D3DX10_IFF_GIF

Graphics Interchange Format (GIF).The file extension for this format is .gif.

D3DX10_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, **D3DX10_IFF_WMP** requires that you initialize COM. Therefore, call **CoInitialize** or **CoInitializeEx** in your application before you call D3DX.

D3DX10_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.

D3DX10 makes use of the Windows Imaging Component to implement the majority of the supported bitmap file types. See Windows Imaging Component Overview for additional information.

Requirements

Requirement	Value
Header	D3DX10Tex.h

See also

D3DX10_MESH enumeration

Article • 04/27/2021 • 2 minutes to read

Flags used to specify creation options for a mesh.

Syntax

```C++
typedef enum D3DX10_MESH {
  D3DX10_MESH_32_BIT        = 0x001,
  D3DX10_MESH_GS_ADJACENCY  = 0x004
} D3DX10_MESH, *LPD3DX10_MESH;
```

Constants

D3DX10_MESH_32_BIT

The mesh has 32-bit indices instead of 16-bit indices. See Remarks.

D3DX10_MESH_GS_ADJACENCY

Signals that the mesh contains geometry shader adjacency data.

Remarks

A 32-bit mesh (D3DXMESH_32BIT) can theoretically support (2^{32})-1 faces and vertices. However, allocating memory for a mesh that large on a 32-bit operating system is not practical.

Requirements

Requirement	Value
Header	D3DX10Mesh.h

See also

D3DX Enumerations

D3DX10_MESH_DISCARD_FLAGS enumeration

Article • 03/15/2021 • 2 minutes to read

Specifies which pieces of mesh data to discard from the device. Used with ID3DX10Mesh::Discard.

Syntax

```C++
typedef enum D3DX10_MESH_DISCARD_FLAGS {
  D3DX10_MESH_DISCARD_ATTRIBUTE_BUFFER  = 0x01,
  D3DX10_MESH_DISCARD_ATTRIBUTE_TABLE   = 0x02,
  D3DX10_MESH_DISCARD_POINTREPS         = 0x04,
  D3DX10_MESH_DISCARD_ADJACENCY         = 0x08,
  D3DX10_MESH_DISCARD_DEVICE_BUFFERS    = 0x10
} D3DX10_MESH_DISCARD_FLAGS, *LPD3DX10_MESH_DISCARD_FLAGS;
```

Constants

D3DX10_MESH_DISCARD_ATTRIBUTE_BUFFER

Discard the attribute buffer.

D3DX10_MESH_DISCARD_ATTRIBUTE_TABLE

Discard the attribute table.

D3DX10_MESH_DISCARD_POINTREPS

Discard the pointer reps buffer.

D3DX10_MESH_DISCARD_ADJACENCY

Discard the adjacency buffer.

D3DX10_MESH_DISCARD_DEVICE_BUFFERS

Discard the buffers committed to the device (with ID3DX10Mesh::CommitToDevice).

Requirements

Requirement	Value
Header	D3DX10Mesh.h

See also

D3DX Enumerations

D3DX10_MESHOPT enumeration

Article • 04/27/2021 • 2 minutes to read

Specifies the type of mesh optimization to be performed.

Syntax

C++

```
typedef enum D3DX10_MESHOPT {
  D3DX10_MESHOPT_COMPACT             = 0x01000000,
  D3DX10_MESHOPT_ATTR_SORT           = 0x02000000,
  D3DX10_MESHOPT_VERTEX_CACHE        = 0x04000000,
  D3DX10_MESHOPT_STRIP_REORDER       = 0x08000000,
  D3DX10_MESHOPT_IGNORE_VERTS        = 0x10000000,
  D3DX10_MESHOPT_DO_NOT_SPLIT        = 0x20000000,
  D3DX10_MESHOPT_DEVICE_INDEPENDENT  = 0x00400000
} D3DX10_MESHOPT, *LPD3DX10_MESHOPT;
```

Constants

D3DX10_MESHOPT_COMPACT

Reorders faces to remove unused vertices and faces.

D3DX10_MESHOPT_ATTR_SORT

Reorders faces to optimize for fewer attribute bundle state changes and enhanced DrawSubset performance.

D3DX10_MESHOPT_VERTEX_CACHE

Reorders faces to increase the cache hit rate of vertex caches.

D3DX10_MESHOPT_STRIP_REORDER

Reorders faces to maximize length of adjacent triangles.

D3DX10_MESHOPT_IGNORE_VERTS

Optimize the faces only; do not optimize the vertices.

D3DX10_MESHOPT_DO_NOT_SPLIT

While attribute sorting, do not split vertices that are shared between attribute groups.

D3DX10_MESHOPT_DEVICE_INDEPENDENT

Affects the vertex cache size. Using this flag specifies a default vertex cache size that works well on legacy hardware.

Remarks

The D3DXMESHOPT_STRIPREORDER and D3DXMESHOPT_VERTEXCACHE optimization flags are mutually exclusive.

The D3DXMESHOPT_SHAREVB flag has been removed from this enumeration. Use D3DXMESH_VB_SHARE instead, in D3DXMESH.

Requirements

Requirement	Value
Header	D3DX10Mesh.h

See also

D3DX Enumerations

D3DX10_NORMALMAP_FLAG enumeration

Article • 03/15/2021 • 2 minutes to read

These flags are used to control how D3DX10ComputeNormalMap generates normal maps. Any number of these flags may be OR'd together in any combination.

Syntax

```C++
typedef enum D3DX10_NORMALMAP_FLAG {
  D3DX10_NORMALMAP_MIRROR_U           = (1 << 16),
  D3DX10_NORMALMAP_MIRROR_V           = (2 << 16),
  D3DX10_NORMALMAP_MIRROR             = (3 << 16),
  D3DX10_NORMALMAP_INVERTSIGN         = (8 << 16),
  D3DX10_NORMALMAP_COMPUTE_OCCLUSION  = (16 << 16)
} D3DX10_NORMALMAP_FLAG, *LPD3DX10_NORMALMAP_FLAG;
```

Constants

D3DX10_NORMALMAP_MIRROR_U

Indicates that pixels off the edge of the texture on the U-axis should be mirrored, not wraped.

D3DX10_NORMALMAP_MIRROR_V

Indicates that pixels off the edge of the texture on the V-axis should be mirrored, not wraped.

D3DX10_NORMALMAP_MIRROR

Same as D3DX10_NORMALMAP_MIRROR_U | D3DX10_NORMALMAP_MIRROR_V.

D3DX10_NORMALMAP_INVERTSIGN

Inverts the direction of each normal.

D3DX10_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.

Requirements

Requirement	Value
Header	D3DX10Tex.h

See also

D3DX Enumerations

D3DX10_SPRITE_FLAG enumeration

Article • 03/15/2021 • 2 minutes to read

Sprite flags that tell the sprite drawing API how to behave. These are passed into ID3DX10Sprite::Begin.

Syntax

C++

```
typedef enum D3DX10_SPRITE_FLAG {
  D3DX10_SPRITE_SORT_TEXTURE              = 0x01,
  D3DX10_SPRITE_SORT_DEPTH_BACK_TO_FRONT  = 0x02,
  D3DX10_SPRITE_SORT_DEPTH_FRONT_TO_BACK  = 0x04,
  D3DX10_SPRITE_SAVE_STATE                = 0x08,
  D3DX10_SPRITE_ADDREF_TEXTURES           = 0x10
} D3DX10_SPRITE_FLAG, *LPD3DX10_SPRITE_FLAG;
```

Constants

D3DX10_SPRITE_SORT_TEXTURE

Sort the sprites by texture before rendering so that when there are many sprites with the same texture that texture all of those sprites will be rendered at the same time, thereby improving performance.

D3DX10_SPRITE_SORT_DEPTH_BACK_TO_FRONT

Sort the sprites from back to front to that those farther away from the camera will be drawn first.

D3DX10_SPRITE_SORT_DEPTH_FRONT_TO_BACK

Sort the sprites from front to back so that those closer to the camera will be drawn first.

D3DX10_SPRITE_SAVE_STATE

Saves the state so that when ID3DX10Sprite::End is called, it will restore the state to the way it was before ID3DX10Sprite::Begin was called.

D3DX10_SPRITE_ADDREF_TEXTURES

Calls AddRef on all of the textures when they are passed into **ID3DX10Sprite::DrawSpritesBuffered**.

Remarks

After a front-to-back or back-to-front sort is done, it will automatically do a secondary sort by texture. This is helpful for when there are many sprites with the same texture all on the same plane, such as when drawing the user interface in a game.

Requirements

Requirement	Value
Header	D3DX10Core.h

See also

D3DX Enumerations

D3DX10_SAVE_TEXTURE_FLAG enumeration

Article • 03/15/2021 • 2 minutes to read

Texture save options.

Syntax

C++

```
typedef enum D3DX10_SAVE_TEXTURE_FLAG {
  D3DX10_STF_USEINPUTBLOB  = 0x0001
} D3DX10_SAVE_TEXTURE_FLAG;
```

Constants

D3DX10_STF_USEINPUTBLOB

Do not optimize.

Requirements

Requirement	Value
Header	D3DX10Tex.h

See also

D3DX Enumerations