

High-level shader language (HLSL)

Article • 08/04/2021 • 2 minutes to read

HLSL is the C-like high-level shader language that you use with programmable shaders in DirectX.

For example, you can use HLSL to write a [vertex shader](#), or a [pixel shader](#), and use those shaders in the implementation of the renderer in your [Direct3D](#) application.

Or you could use HLSL to write a compute shader, perhaps to implement a physics simulation. However, if for example you're inclined to write your own convolution operator (for image processing) as HLSL in a compute shader, then you'll get better performance in that scenario if you use [Direct Machine Learning \(DirectML\)](#) instead.

HLSL was created (starting with DirectX 9) to set up the programmable 3D [pipeline](#). You can program the entire pipeline with HLSL instructions.

Where to go next

- [Programming guide for HLSL](#)
- [Reference for HLSL](#)

Programming guide for HLSL

For a conceptual introduction to HLSL, see the [Programming guide for HLSL](#).

The programming guide discusses the different kinds of shader stages, and how to create, compile, optimize, bind, and link shaders.

There you'll also find overviews of, and release notes about, the successive generations of shader model version that have been released, going back as far as HLSL shader model 5.

Reference for HLSL

For HLSL reference documentation, see the [Reference for HLSL](#).

The reference section has a complete listing of the language syntax and of the intrinsic functions that are built into HLSL in order to simplify your coding requirements.

There also you'll find a discussion of shader models versus profiles, and shader model reference content going back as far as HLSL shader model 1. There's also content

covering assembly instructions, the D3DCompiler tool, and info about the errors and warnings that a shader can return.

Programming guide for HLSL

Article • 01/12/2021 • 2 minutes to read

Data enters the graphics pipeline as a stream of primitives and is processed by the shader stages. The actual shader stages depend on the version of Direct3D, but certainly include the vertex, pixel and geometry stages. Other stages include the hull and domain shaders for tessellation, and the compute shader. These stages are completely programmable using the High Level Shading Language ([HLSL](#)).

HLSL shaders can be compiled at author-time or at runtime, and set at runtime into the appropriate pipeline stage. Direct3D 9 shaders can be designed using [shader model 1](#), [shader model 2](#) and [shader model 3](#); Direct3D 10 shaders can only be designed on [shader model 4](#). Direct3D 11 shaders can be designed on [shader model 5](#). Direct3D 11.3 and Direct3D 12 can be designed on [shader model 5.1](#), and Direct3D 12 can also be designed on [shader model 6](#).

In this section

Topic	Description
Using shader linking	We show how to create precompiled HLSL functions, package them into libraries, and link them into full shaders at run-time.
Writing HLSL Shaders in Direct3D 9	
Using Shaders in Direct3D 9	
Using Shaders in Direct3D 10	
Optimizing HLSL Shaders	
Debugging Shaders in Visual Studio	The latest tool for debugging shaders now ships as a feature in Microsoft Visual Studio, called Visual Studio Graphics Debugger.
Compiling Shaders	Let's now look at various ways to compile your shader code and conventions for file extensions for shader code.
Specifying Compiler Targets	Here we list the targets for various profiles that the D3DCompile* functions and the HLSL compiler support.

Topic	Description
Unpacking and Packing DXGI_FORMAT for In-Place Image Editing	
Using HLSL minimum precision	Starting with Windows 8, graphics drivers can implement minimum precision HLSL scalar data types by using any precision greater than or equal to their specified bit precision.
HLSL Shader Model 5	
HLSL Shader Model 5.1	This section describes the features of Shader Model 5.1 as they apply in practice to D3D12 and D3D11.3. All DirectX 12 hardware supports Shader Model 5.1.
HLSL Shader Model 6.0	Describes the wave operation intrinsics added to HLSL Shader Model 6.0.
HLSL Shader Model 6.4	Describes the machine learning intrinsics added to HLSL Shader Model 6.4.

Related topics

- [HLSL](#)
- [Reference for HLSL](#)

Using shader linking

Article • 08/19/2020 • 2 minutes to read

We show how to create precompiled HLSL functions, package them into libraries, and link them into full shaders at run-time. Shader linking is supported starting with Windows 8.1.

Objective: Learn how to use shader linking.

Prerequisites

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

Total time to complete: 60 minutes.

Where to go from here

Also see [HLSL compiler APIs](#).

We show you how to:

- Compile your shader code
- Load the compiled code into a shader library
- Bind the resources from source slots to destination slots
- Construct function-linking-graphs (FLGs) for shaders
- Link shader graphs with a shader library to produce a shader blob that the Direct3D runtime can use

Next we make a shader library and bind resources from source slots to destination slots.

[Packaging a shader library](#)

Related topics

[Programming Guide for HLSL](#)

[Direct3D 11 Graphics](#)

[DXGI](#)

Packaging a shader library

Article • 08/19/2020 • 2 minutes to read

Here we show you how to compile your shader code, load the compiled code into a shader library, and bind resources from source slots to destination slots.

Objective: To package a shader library to use for shader linking.

Prerequisites

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

Time to complete: 30 minutes.

Instructions

1. Compiling your shader code

Compile your shader code with one of the compile functions. For example, this code snippet uses [D3DCompile](#).

```
string source;

ComPtr<ID3DBlob> codeBlob;
ComPtr<ID3DBlob> errorBlob;
HRESULT hr = D3DCompile(
    source.c_str(),
    source.size(),
    "ShaderModule",
    NULL,
    NULL,
    NULL,
    ("lib" + m_shaderModelSuffix).c_str(),
    D3DCOMPILE_OPTIMIZATION_LEVEL3,
    0,
    &codeBlob,
    &errorBlob
);
```

The source string contains the uncompiled ASCII HLSL code.

2. Load the compiled code into a shader library.

Call the [D3DLoadModule](#) function to load the compiled code ([ID3DBlob](#)) into a module ([ID3D11Module](#)) that represents a shader library.

```
// Load the compiled library code into a module object.  
ComPtr<ID3D11Module> shaderLibrary;  
DX::ThrowIfFailed(D3DLoadModule(codeBlob->GetBufferPointer(), codeBlob-  
>GetBufferSize(), &shaderLibrary));
```

3. Bind resources from source slots to destination slots.

Call the [ID3D11Module::CreateInstance](#) method to create an instance ([ID3D11ModuleInstance](#)) of the library so you can then define resource bindings for the instance.

Call the bind methods of [ID3D11ModuleInstance](#) to bind the resources you need from source slots to destination slots. The resources can be textures, buffers, samplers, constant buffers, or UAVs. Typically, you will use the same slots as the source library.

```
// Create an instance of the library and define resource bindings for  
it.  
// In this sample we use the same slots as the source library however  
this is not required.  
ComPtr<ID3D11ModuleInstance> shaderLibraryInstance;  
DX::ThrowIfFailed(shaderLibrary->CreateInstance("",  
&shaderLibraryInstance));  
// HRESULTs for Bind methods are intentionally ignored as compiler  
optimizations may eliminate the source  
// bindings. In these cases, the Bind operation will fail, but the final  
shader will function normally.  
shaderLibraryInstance->BindResource(0, 0, 1);  
shaderLibraryInstance->BindSampler(0, 0, 1);  
shaderLibraryInstance->BindConstantBuffer(0, 0, 0);  
shaderLibraryInstance->BindConstantBuffer(1, 1, 0);  
shaderLibraryInstance->BindConstantBuffer(2, 2, 0);
```

This HLSL code shows that the source library uses the same slots (t0, s0, b0, b1, and b2) as the slots used in the preceding bind methods of [ID3D11ModuleInstance](#).

```
// This is the default code in the fixed header section.  
Texture2D<float3> Texture : register(t0);  
SamplerState Anisotropic : register(s0);  
cbuffer CameraData : register(b0)  
{  
    float4x4 Model;  
    float4x4 View;  
    float4x4 Projection;  
};  
cbuffer TimeVariantSignals : register(b1)  
{  
    float SineWave;  
    float SquareWave;  
    float TriangleWave;  
    float SawtoothWave;  
};  
  
// This code is not displayed, but is used as part of the linking process.  
cbuffer HiddenBuffer : register(b2)  
{  
    float3 LightDirection;  
};
```

Summary and next steps

We compiled shader code, loaded the compiled code into a shader library, and bound resources from source slots to destination slots.

Next we construct function-linking-graphs (FLGs) for shaders, link them to compiled code, and produce shader blobs that the Direct3D runtime can use.

[Constructing a function-linking-graph and linking it to compiled code](#)

Related topics

[Using shader linking](#)

Constructing a function-linking-graph and linking it to compiled code

Article • 08/19/2020 • 7 minutes to read

Here we show you how to construct function-linking-graphs (FLGs) for shaders and how to link those shaders with a shader library to produce shader blobs that the Direct3D runtime can use.

Objective: To construct a function-linking-graph and link it to compiled code.

Prerequisites

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

We also assume that you went through [Packaging a shader library](#).

Time to complete: 30 minutes.

Instructions

1. Construct a function-linking-graph for the vertex shader.

Call the [D3DCreateFunctionLinkingGraph](#) function to create a function-linking-graph ([ID3D11FunctionLinkingGraph](#)) to represent the vertex shader.

Use an array of [D3D11_PARAMETER_DESC](#) structures to define the input parameters for the vertex shader. The [Input-Assembler Stage](#) feeds the input parameters to the vertex shader. The layout of the vertex shader's input parameters matches the layout of the vertex shader in the compiled code. After you define the input parameters, call the [ID3D11FunctionLinkingGraph::SetInputSignature](#) method to define the input node ([ID3D11LinkingNode](#)) for the vertex shader.

C++

```
ComPtr<ID3D11FunctionLinkingGraph> vertexShaderGraph;
DX::ThrowIfFailed(D3DCreateFunctionLinkingGraph(0,
&vertexShaderGraph));

// Define the main input node which will be fed by the Input
```

```

Assembler pipeline stage.
    static const D3D11_PARAMETER_DESC vertexShaderInputParameters[] =
    {
        {"inputPos", "POSITION0", D3D_SVT_FLOAT, D3D_SVC_VECTOR, 1,
3, D3D_INTERPOLATION_LINEAR, D3D_PF_IN, 0, 0, 0, 0},
        {"inputTex", "TEXCOORD0", D3D_SVT_FLOAT, D3D_SVC_VECTOR, 1,
2, D3D_INTERPOLATION_LINEAR, D3D_PF_IN, 0, 0, 0, 0},
        {"inputNorm", "NORMAL0", D3D_SVT_FLOAT, D3D_SVC_VECTOR, 1,
3, D3D_INTERPOLATION_LINEAR, D3D_PF_IN, 0, 0, 0, 0}
    };
    ComPtr<ID3D11LinkingNode> vertexShaderInputNode;
    LinkingThrowIfFailed(vertexShaderGraph-
>SetInputSignature(vertexShaderInputParameters,
ARRAYSIZE(vertexShaderInputParameters),
&vertexShaderInputNode), vertexShaderGraph.Get());

```

Call the [ID3D11FunctionLinkingGraph::CallFunction](#) method to create a node for the main vertex shader function and make calls to [ID3D11FunctionLinkingGraph::PassValue](#) to pass values from the input node to the node for the main vertex shader function.

C++

```

// Create a node for the main VertexFunction call using the
output of the helper functions.
ComPtr<ID3D11LinkingNode> vertexFunctionCallNode;
LinkingThrowIfFailed(vertexShaderGraph->CallFunction(
"", shaderLibrary.Get(), "VertexFunction", &vertexFunctionCallNode),
vertexShaderGraph.Get());

// Define the graph edges from the input node and helper
function nodes.
LinkingThrowIfFailed(vertexShaderGraph-
>PassValue(homogenizeCallNodeForPos.Get(), D3D_RETURN_PARAMETER_INDEX,
vertexFunctionCallNode.Get(), 0), vertexShaderGraph.Get());
LinkingThrowIfFailed(vertexShaderGraph-
>PassValue(vertexShaderInputNode.Get(), 1, vertexFunctionCallNode.Get(), 1),
vertexShaderGraph.Get());
LinkingThrowIfFailed(vertexShaderGraph-
>PassValue(homogenizeCallNodeForNorm.Get(), D3D_RETURN_PARAMETER_INDEX,
vertexFunctionCallNode.Get(), 2), vertexShaderGraph.Get());

```

Use an array of [D3D11_PARAMETER_DESC](#) structures to define the output parameters for the vertex shader. The vertex shader feeds its output parameters to the pixel shader. The layout of the vertex shader's output parameters matches the layout of the pixel shader in the compiled code. After you define the output parameters, call the [ID3D11FunctionLinkingGraph::SetOutputSignature](#) method to define the output node ([ID3D11LinkingNode](#)) for the vertex shader.

C++

```
// Define the main output node which will feed the Pixel Shader
pipeline stage.
static const D3D11_PARAMETER_DESC vertexShaderOutputParameters[]
=
{
    {"outputTex", "TEXCOORD0", D3D_SVT_FLOAT, D3D_SVC_VECTOR,
1, 2, D3D_INTERPOLATION_UNDEFINED, D3D_PF_OUT, 0, 0, 0, 0},
    {"outputNorm", "NORMAL0", D3D_SVT_FLOAT, D3D_SVC_VECTOR,
1, 3, D3D_INTERPOLATION_UNDEFINED, D3D_PF_OUT, 0, 0, 0, 0},
    {"outputPos", "SV_POSITION", D3D_SVT_FLOAT, D3D_SVC_VECTOR,
1, 4, D3D_INTERPOLATION_UNDEFINED, D3D_PF_OUT, 0, 0, 0, 0}
};
ComPtr<ID3D11LinkingNode> vertexShaderOutputNode;
LinkingThrowIfFailed(vertexShaderGraph-
>SetOutputSignature(vertexShaderOutputParameters,
ARRAYSIZE(vertexShaderOutputParameters),
&vertexShaderOutputNode), vertexShaderGraph.Get());
```

Make calls to [ID3D11FunctionLinkingGraph::PassValue](#) to pass values from the node for the main vertex shader function to the output node.

C++

```
LinkingThrowIfFailed(vertexShaderGraph-
>PassValue(vertexFunctionCallNode.Get(), 0, vertexShaderOutputNode.Get(),
2),
vertexShaderGraph.Get());
LinkingThrowIfFailed(vertexShaderGraph-
>PassValue(vertexFunctionCallNode.Get(), 1, vertexShaderOutputNode.Get(),
0),
vertexShaderGraph.Get());
LinkingThrowIfFailed(vertexShaderGraph-
>PassValue(vertexFunctionCallNode.Get(), 2, vertexShaderOutputNode.Get(),
1),
vertexShaderGraph.Get());
```

Call the [ID3D11FunctionLinkingGraph::CreateModuleInstance](#) method to finalize the vertex shader graph.

C++

```
// Finalize the vertex shader graph.
ComPtr<ID3D11ModuleInstance> vertexShaderGraphInstance;
LinkingThrowIfFailed(vertexShaderGraph-
>CreateModuleInstance(&vertexShaderGraphInstance, nullptr),
vertexShaderGraph.Get());
```

2. Link the vertex shader

Call the [D3DCreateLinker](#) function to create a linker ([ID3D11Linker](#)) that you can use to link the instance of the shader library that you created in [Packaging a shader library](#) with the instance of the vertex shader graph that you created in the preceding step. Call the [ID3D11Linker::UseLibrary](#) method to specify the shader library to use for linking. Call the [ID3D11Linker::Link](#) method to link the shader library with the vertex shader graph and to produce a pointer to the [ID3DBlob](#) interface that you can use to access the compiled vertex shader code. You can then pass this compiled vertex shader code to the [ID3D11Device::CreateVertexShader](#) method to create the vertex shader object and to the [ID3D11Device::CreateInputLayout](#) method to create the input-layout object.

C++

```
// Create a linker and hook up the module instance.
ComPtr<ID3D11Linker> linker;
DX::ThrowIfFailed(D3DCreateLinker(&linker));
DX::ThrowIfFailed(linker-
>UseLibrary(shaderLibraryInstance.Get()));

// Link the vertex shader.
ComPtr<ID3DBlob> errorBlob;
if (FAILED(linker->Link(vertexShaderGraphInstance.Get(), "main",
("vs" + m_shaderModelSuffix).c_str(), 0, &vertexShaderBlob,
&errorBlob)))
{
    throw errorBlob;
}

ComPtr<ID3D11VertexShader> vertexShader;
DX::ThrowIfFailed(
    device->CreateVertexShader(
        vertexShaderBlob->GetBufferPointer(),
        vertexShaderBlob->GetBufferSize(),
        nullptr,
        &vertexShader
    )
);
context->VSSetShader(vertexShader.Get(), nullptr, 0);
D3D11_INPUT_ELEMENT_DESC inputLayoutDesc[] =
{
    { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 12,
D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { "NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 20,
D3D11_INPUT_PER_VERTEX_DATA, 0 }
};
ComPtr<ID3D11InputLayout> inputLayout;
DX::ThrowIfFailed(device->CreateInputLayout(inputLayoutDesc,
```

```
ARRAYSIZE(inputLayoutDesc), vertexShaderBlob->GetBufferPointer(),
vertexShaderBlob->GetBufferSize(), &inputLayout));
context->IASetInputLayout(inputLayout.Get());
```

3. Construct a function-linking-graph for the pixel shader.

Call the [D3DCreateFunctionLinkingGraph](#) function to create a function-linking-graph ([ID3D11FunctionLinkingGraph](#)) to represent the pixel shader.

Use an array of [D3D11_PARAMETER_DESC](#) structures to define the input parameters for the pixel shader. The vertex shader stage feeds the input parameters to the pixel shader. The layout of the pixel shader's input parameters matches the layout of the pixel shader in the compiled code. After you define the input parameters, call the [ID3D11FunctionLinkingGraph::SetInputSignature](#) method to define the input node ([ID3D11LinkingNode](#)) for the pixel shader.

C++

```
ComPtr<ID3D11FunctionLinkingGraph> pixelShaderGraph;
DX::ThrowIfFailed(D3DCreateFunctionLinkingGraph(0,
&pixelShaderGraph));

// Define the main input node which will be fed by the vertex
// shader pipeline stage.
static const D3D11_PARAMETER_DESC pixelShaderInputParameters[] =
{
    {"inputTex", "TEXCOORD0", D3D_SVT_FLOAT, D3D_SVC_VECTOR,
1, 2, D3D_INTERPOLATION_UNDEFINED, D3D_PF_IN, 0, 0, 0, 0},
    {"inputNorm", "NORMAL0", D3D_SVT_FLOAT, D3D_SVC_VECTOR,
1, 3, D3D_INTERPOLATION_UNDEFINED, D3D_PF_IN, 0, 0, 0, 0},
    {"inputPos", "SV_POSITION", D3D_SVT_FLOAT, D3D_SVC_VECTOR,
1, 4, D3D_INTERPOLATION_UNDEFINED, D3D_PF_IN, 0, 0, 0, 0}
};
ComPtr<ID3D11LinkingNode> pixelShaderInputNode;
LinkingThrowIfFailed(pixelShaderGraph-
>SetInputSignature(pixelShaderInputParameters,
ARRAYSIZE(pixelShaderInputParameters),
&pixelShaderInputNode), pixelShaderGraph.Get());
```

Call the [ID3D11FunctionLinkingGraph::CallFunction](#) method to create a node for the main pixel shader function and make calls to [ID3D11FunctionLinkingGraph::PassValue](#) to pass values from the input node to the node for the main pixel shader function.

C++

```
// Create a node for the main ColorFunction call and connect it
// to the pixel shader inputs.
ComPtr<ID3D11LinkingNode> colorValueNode;
```

```

        LinkingThrowIfFailed(pixelShaderGraph->CallFunction("",  

shaderLibrary.Get(), "ColorFunction", &colorValueNode),  

pixelShaderGraph.Get());  
  

        // Define the graph edges from the input node.  

        LinkingThrowIfFailed(pixelShaderGraph-  

>PassValue(pixelShaderInputNode.Get(), 0, colorValueNode.Get(), 0),  

pixelShaderGraph.Get());  

        LinkingThrowIfFailed(pixelShaderGraph-  

>PassValue(pixelShaderInputNode.Get(), 1, colorValueNode.Get(), 1),  

pixelShaderGraph.Get());
```

Use an array of [D3D11_PARAMETER_DESC](#) structures to define the output parameters for the pixel shader. The pixel shader feeds its output parameters to the [Output-Merger Stage](#). After you define the output parameters, call the [ID3D11FunctionLinkingGraph::SetOutputSignature](#) method to define the output node ([ID3D11LinkingNode](#)) for the pixel shader and make calls to [ID3D11FunctionLinkingGraph::PassValue](#) to pass values from a pixel shader function node to the output node.

C++

```

// Define the main output node which will feed the Output Merger
pipeline stage.
D3D11_PARAMETER_DESC pixelShaderOutputParameters[] =
{
    {"outputColor", "SV_TARGET", D3D_SVT_FLOAT, D3D_SVC_VECTOR,
1, 4, D3D_INTERPOLATION_UNDEFINED, D3D_PF_OUT, 0, 0, 0, 0}
};
ComPtr<ID3D11LinkingNode> pixelShaderOutputNode;
LinkingThrowIfFailed(pixelShaderGraph-
>SetOutputSignature(pixelShaderOutputParameters,
ARRAYSIZE(pixelShaderOutputParameters),
&pixelShaderOutputNode), pixelShaderGraph.Get());
LinkingThrowIfFailed(pixelShaderGraph-
>PassValue(fillAlphaCallNode.Get(), D3D_RETURN_PARAMETER_INDEX,
pixelShaderOutputNode.Get(), 0),
pixelShaderGraph.Get());
```

Call the [ID3D11FunctionLinkingGraph::CreateModuleInstance](#) method to finalize the pixel shader graph.

C++

```

// Finalize the pixel shader graph.
ComPtr<ID3D11ModuleInstance> pixelShaderGraphInstance;
LinkingThrowIfFailed(pixelShaderGraph-
>CreateModuleInstance(&pixelShaderGraphInstance, nullptr),
pixelShaderGraph.Get());
```

4. Link the pixel shader

Call the [D3DCreateLinker](#) function to create a linker ([ID3D11Linker](#)) that you can use to link the instance of the shader library that you created in [Packaging a shader library](#) with the instance of the pixel shader graph that you created in the preceding step. Call the [ID3D11Linker::UseLibrary](#) method to specify the shader library to use for linking. Call the [ID3D11Linker::Link](#) method to link the shader library with the pixel shader graph and to produce a pointer to the [ID3DBlob](#) interface that you can use to access the compiled pixel shader code. You can then pass this compiled pixel shader code to the [ID3D11Device::CreatePixelShader](#) method to create the pixel shader object.

C++

```
// Create a linker and hook up the module instance.
ComPtr<ID3D11Linker> linker;
DX::ThrowIfFailed(D3DCreateLinker(&linker));
DX::ThrowIfFailed(linker-
>UseLibrary(shaderLibraryInstance.Get()));

// Link the pixel shader.
ComPtr<ID3DBlob> errorBlob;
if (FAILED(linker->Link(pixelShaderGraphInstance.Get(), "main",
("ps" + m_shaderModelSuffix).c_str(), 0, &pixelShaderBlob, &errorBlob)))
{
    throw errorBlob;
}

ComPtr<ID3D11PixelShader> pixelShader;
DX::ThrowIfFailed(
    device->CreatePixelShader(
        pixelShaderBlob->GetBufferPointer(),
        pixelShaderBlob->GetBufferSize(),
        nullptr,
        &pixelShader
    )
);
context->PSSetShader(pixelShader.Get(), nullptr, 0);
```

Summary

We used the [ID3D11FunctionLinkingGraph](#) methods to construct the vertex and pixel shader graphs and to specify the shader structure programmatically.

These graph constructions consist of sequences of precompiled function calls that pass values to each other. FLG nodes ([ID3D11LinkingNode](#)) represent input and output shader nodes and invocations of precompiled library functions. The order in which you

register the function-call nodes defines the sequence of invocations. You must specify the input node ([ID3D11FunctionLinkingGraph::SetInputSignature](#)) first and the output node last ([ID3D11FunctionLinkingGraph::SetOutputSignature](#)). FLG edges define how values are passed from one node to another. The data types of passed values must be the same; there is no implicit type conversion. Shape and swizzling rules follow the HLSL behavior. Values can only be passed forward in this sequence.

We also used [ID3D11Linker](#) methods to link the shader library with the shader graphs and to produce shader blobs for the Direct3D runtime to use.

Congratulations! You are now ready to use shader linking in your own apps.

Related topics

[Using shader linking](#)

Writing HLSL Shaders in Direct3D 9

Article • 05/24/2021 • 22 minutes to read

- [Vertex-Shader Basics](#)
- [Pixel-Shader Basics](#)
 - [Texture Stage and Sampler States](#)
 - [Pixel Shader Inputs](#)
 - [Pixel Shader Outputs](#)
- [Shader Inputs and Shader Variables](#)
 - [Declaring Shader Variables](#)
 - [Uniform Shader Inputs](#)
 - [Varying Shader Inputs and Semantics](#)
 - [Samplers and Texture Objects](#)
- [Writing Functions](#)
- [Flow Control](#)
- [Related topics](#)

Vertex-Shader Basics

When in operation, a programmable vertex shader replaces the vertex processing done by the Microsoft Direct3D graphics pipeline. While using a vertex shader, state information regarding transformation and lighting operations is ignored by the fixed function pipeline. When the vertex shader is disabled and fixed function processing is returned, all current state settings apply.

Tessellation of high-order primitives should be done before the vertex shader executes. Implementations that perform surface tessellation after the shader processing must do so in a way that is not apparent to the application and shader code.

As a minimum, a vertex shader must output vertex position in homogeneous clip space. Optionally, the vertex shader can output texture coordinates, vertex color, vertex lighting, fog factors, and so on.

Pixel-Shader Basics

Pixel processing is performed by pixel shaders on individual pixels. Pixel shaders work in concert with vertex shaders; the output of a vertex shader provides the inputs for a pixel shader. Other pixel operations (fog blending, stencil operations, and render-target blending) occur after execution of the shader.

Texture Stage and Sampler States

A pixel shader completely replaces the pixel-blending functionality specified by the multi-texture blender including operations previously defined by the texture stage states. Texture sampling and filtering operations which were controlled by the standard texture stage states for minification, magnification, mip filtering, and the wrap addressing modes, can be initialized in shaders. The application is free to change these states without requiring the regeneration of the currently bound shader. Setting state can be made even easier if your shaders are designed within an effect.

Pixel Shader Inputs

For pixel shader versions ps_1_1 - ps_2_0, diffuse and specular colors are saturated (clamped) in the range 0 to 1 before use by the shader.

Color values input to the pixel shader are assumed to be perspective correct, but this is not guaranteed (for all hardware). Colors sampled from texture coordinates are iterated in a perspective correct manner, and are clamped to the 0 to 1 range during iteration.

Pixel Shader Outputs

For pixel shader versions ps_1_1 - ps_1_4, the result emitted by the pixel shader is the contents of register r0. Whatever it contains when the shader completes processing is sent to the fog stage and render-target blender.

For pixel shader versions ps_2_0 and above, output color is emitted from oC0 - oC4.

Shader Inputs and Shader Variables

- [Declaring Shader Variables](#)
- [Uniform Shader Inputs](#)
- [Varying Shader Inputs and Semantics](#)
- [Samplers and Texture Objects](#)

Declaring Shader Variables

The simplest variable declaration includes a type and a variable name, such as this floating-point declaration:

```
float fVar;
```

You can initialize a variable in the same statement.

```
float fVar = 3.1f;
```

An array of variables can be declared,

```
int iVar[3];
```

or declared and initialized in the same statement.

```
int iVar[3] = {1,2,3};
```

Here are a few declarations that demonstrate many of the characteristics of high-level shader language (HLSL) variables:

```
float4 color;
uniform float4 position : POSITION;
const float4 lightDirection = {0,0,1};
```

Data declarations can use any valid type including:

- [Data Types \(DirectX HLSL\)](#)
- [Vector Type \(DirectX HLSL\)](#)
- [Matrix Type \(DirectX HLSL\)](#)
- [Shader Type \(DirectX HLSL\)](#)
- [Sampler Type \(DirectX HLSL\)](#)
- [User-Defined Type \(DirectX HLSL\)](#)

A shader can have top-level variables, arguments, and functions.

```
// top-level variable
float globalShaderVariable;
```

```
// top-level function
void function(
    in float4 position: POSITION0 // top-level argument
)
{
    float localShaderVariable; // local variable
    function2(...)
}

void function2()
{
    ...
}
```

Top-level variables are declared outside of all functions. Top-level arguments are parameters to a top-level function. A top-level function is any function called by the application (as opposed to a function that is called by another function).

Uniform Shader Inputs

Vertex and pixel shaders accept two kinds of input data: varying and uniform. The varying input is the data that is unique to each execution of the shader. For a vertex shader, the varying data (for example: position, normal, etc.) comes from the vertex streams. The uniform data (for example: material color, world transform, etc.) is constant for multiple executions of a shader. For those familiar with the assembly shader models, uniform data is specified by constant registers and varying data by the v and t registers.

Uniform data can be specified by two methods. The most common method is to declare global variables and use them within a shader. Any use of global variables within a shader will result in adding that variable to the list of uniform variables required by that shader. The second method is to mark an input parameter of the top-level shader function as uniform. This marking specifies that the given variable should be added to the list of uniform variables.

Uniform variables used by a shader are communicated back to the application via the constant table. The constant table is the name for the symbol table that defines how the uniform variables used by a shader fit into the constant registers. The uniform function parameters appear in the constant table prepended with a dollar sign (\$), unlike the global variables. The dollar sign is required to avoid name collisions between local uniform inputs and global variables of the same name.

The constant table contains the constant register locations of all uniform variables used by the shader. The table also includes the type information and the default value, if specified.

Varying Shader Inputs and Semantics

Varying input parameters (of a top-level shader function) must be marked either with a semantic or uniform keyword indicating the value is constant for the execution of the shader. If a top-level shader input is not marked with a semantic or uniform keyword, then the shader will fail to compile.

The input semantic is a name used to link the given input to an output of the previous part of the graphics pipeline. For example, the input semantic POSITION0 is used by the vertex shaders to specify where the position data from the vertex buffer should be linked.

Pixel and vertex shaders have different sets of input semantics due to the different parts of the graphics pipeline that feed into each shader unit. Vertex shader input semantics describe the per-vertex information (for example: position, normal, texture coordinates, color, tangent, binormal, etc.) to be loaded from a vertex buffer into a form that can be consumed by the vertex shader. The input semantics directly map to the vertex declaration usage and the usage index.

Pixel shader input semantics describe the information that is provided per pixel by the rasterization unit. The data is generated by interpolating between outputs of the vertex shader for each vertex of the current primitive. The basic pixel shader input semantics link the output color and texture coordinate information to input parameters.

Input semantics can be assigned to shader input by two methods:

- Appending a colon and the semantic name to the parameter declaration.
- Defining an input structure with input semantics assigned to each structure member.

Vertex and pixel shaders provide output data to the subsequent graphics pipeline stage. Output semantics are used to specify how data generated by the shader should be linked to the inputs of the next stage. For example, the output semantics for a vertex shader are used to link the outputs of the interpolators in the rasterizer to generate the input data for the pixel shader. The pixel shader outputs are the values provided to the alpha blending unit for each of the render targets or the depth value written to the depth buffer.

Vertex shader output semantics are used to link the shader both to the pixel shader and to the rasterizer stage. A vertex shader that is consumed by the rasterizer and not exposed to the pixel shader must generate position data as a minimum. Vertex shaders that generate texture coordinate and color data provide that data to a pixel shader after interpolation is done.

Pixel shader output semantics bind the output colors of a pixel shader with the correct render target. The pixel shader output color is linked to the alpha blend stage, which determines how the destination render targets are modified. The pixel shader depth output can be used to change the destination depth values at the current raster location. The depth output and multiple render targets are only supported with some shader models.

The syntax for output semantics is identical to the syntax for specifying input semantics. The semantics can be either specified directly on parameters declared as "out" parameters or assigned during the definition of a structure that either returned as an "out" parameter or the return value of a function.

Semantics identify where data comes from. Semantics are optional identifiers that identify shader inputs and outputs. Semantics appear in one of three places:

- After a structure member.
- After an argument in a function's input argument list.
- After the function's input argument list.

This example uses a structure to provide one or more vertex shader inputs, and another structure to provide one or more vertex shader outputs. Each of the structure members uses a semantic.

```
vector vClr;

struct VS_INPUT
{
    float4 vPosition : POSITION;
    float3 vNormal : NORMAL;
    float4 vBlendWeights : BLENDWEIGHT;
};

struct VS_OUTPUT
{
    float4 vPosition : POSITION;
    float4 vDiffuse : COLOR;
};

float4x4 mWld1;
float4x4 mWld2;
float4x4 mWld3;
float4x4 mWld4;

float Len;
float4 vLight;
```

```

float4x4 mTot;

VS_OUTPUT VS_Skinning_Example(const VS_INPUT v, uniform float len=100)
{
    VS_OUTPUT out;

    // Skin position (to world space)
    float3 vPosition =
        mul(v.vPosition, (float4x3) mWld1) * v.vBlendWeights.x +
        mul(v.vPosition, (float4x3) mWld2) * v.vBlendWeights.y +
        mul(v.vPosition, (float4x3) mWld3) * v.vBlendWeights.z +
        mul(v.vPosition, (float4x3) mWld4) * v.vBlendWeights.w;
    // Skin normal (to world space)
    float3 vNormal =
        mul(v.vNormal, (float3x3) mWld1) * v.vBlendWeights.x +
        mul(v.vNormal, (float3x3) mWld2) * v.vBlendWeights.y +
        mul(v.vNormal, (float3x3) mWld3) * v.vBlendWeights.z +
        mul(v.vNormal, (float3x3) mWld4) * v.vBlendWeights.w;

    // Output stuff
    out.vPosition = mul(float4(vPosition + vNormal * Len, 1), mTot);
    out.vDiffuse = dot(vLight, vNormal);

    return out;
}

```

The input structure identifies the data from the vertex buffer that will provide the shader inputs. This shader maps the data from the position, normal, and blendweight elements of the vertex buffer into vertex shader registers. The input data type does not have to exactly match the vertex declaration data type. If it doesn't exactly match, the vertex data will automatically be converted into the HLSL's data type when it is written into the shader registers. For instance, if the normal data were defined to be of type `UINT` by the application, it would be converted into a `float3` when read by the shader.

If the data in the vertex stream contains fewer components than the corresponding shader data type, the missing components will be initialized to 0 (except for w, which is initialized to 1).

Input semantics are similar to the values in the [D3DDECLUSAGE](#).

The output structure identifies the vertex shader output parameters of position and color. These outputs will be used by the pipeline for triangle rasterization (in primitive processing). The output marked as position data denotes the position of a vertex in homogeneous space. As a minimum, a vertex shader must generate position data. The screen space position is computed after the vertex shader completes by dividing the (x, y, z) coordinate by w. In screen space, -1 and 1 are the minimum and maximum x and y values of the boundaries of the viewport, while z is used for z-buffer testing.

Output semantics are also similar to the values in [D3DDECLUSAGE](#). In general, an output structure for a vertex shader can also be used as the input structure for a pixel shader, provided the pixel shader does not read from any variable marked with the position, point size, or fog semantics. These semantics are associated with per-vertex scalar values that are not used by a pixel shader. If these values are needed for the pixel shader, they can be copied into another output variable that uses a pixel shader semantic.

Global variables are assigned to registers automatically by the compiler. Global variables are also called uniform parameters because the contents of the variable is the same for all pixels processed each time the shader is called. The registers are contained in the constant table, which can be read using the [ID3DXConstantTable](#) interface.

Input semantics for pixel shaders map values into specific hardware registers for transport between vertex shaders and pixel shaders. Each register type has specific properties. Because there are currently only two semantics for color and texture coordinates, it is common for most data to be marked as a texture coordinate even when it is not.

Notice that the vertex shader output structure used an input with position data, which is not used by the pixel shader. HLSL allows valid output data of a vertex shader that is not valid input data for a pixel shader, provided that it is not referenced in the pixel shader.

Input arguments can also be arrays. Semantics are automatically incremented by the compiler for each element of the array. For instance, consider the following explicit declaration:

```
struct VS_OUTPUT
{
    float4 Position    : POSITION;
    float3 Diffuse     : COLOR0;
    float3 Specular    : COLOR1;
    float3 HalfVector  : TEXCOORD3;
    float3 Fresnel      : TEXCOORD2;
    float3 Reflection   : TEXCOORD0;
    float3 NoiseCoord   : TEXCOORD1;
};

float4 Sparkle(VS_OUTPUT In) : COLOR
```

The explicit declaration given above is equivalent to the following declaration that will have semantics automatically incremented by the compiler:

```
float4 Sparkle(float4 Position : POSITION,
               float3 Col[2] : COLOR0,
               float3 Tex[4] : TEXCOORD0) : COLOR0
{
    // shader statements
    ...
}
```

Just like input semantics, output semantics identify data usage for pixel shader output data. Many pixel shaders write to only one output color. Pixel shaders can also write out a depth value into one or more multiple render targets at the same time (up to four). Like vertex shaders, pixel shaders use a structure to return more than one output. This shader writes 0 to the color components, as well as to the depth component.

```
struct PS_OUTPUT
{
    float4 Color[4] : COLOR0;
    float Depth : DEPTH;
};

PS_OUTPUT main(void)
{
    PS_OUTPUT out;

    // Shader statements
    ...

    // Write up to four pixel shader output colors
    out.Color[0] = ...
    out.Color[1] = ...
    out.Color[2] = ...
    out.Color[3] = ...

    // Write pixel depth
    out.Depth = ...

    return out;
}
```

Pixel shader output colors must be of type `float4`. When writing multiple colors, all output colors must be used contiguously. In other words, `COLOR1` cannot be an output unless `COLOR0` has already been written. Pixel shader depth output must be of type `float1`.

Samplers and Texture Objects

A sampler contains sampler state. Sampler state specifies the texture to be sampled, and controls the filtering that is done during sampling. Three things are required to sample a texture:

- A texture
- A sampler (with sampler state)
- A sampling instruction

Samplers can be initialized with textures and sampler state as shown here:

```
sampler s = sampler_state
{
    texture = NULL;
    mipfilter = LINEAR;
};
```

Here's an example of the code to sample a 2D texture:

```
texture tex0;
sampler2D s_2D;

float2 sample_2D(float2 tex : TEXCOORD0) : COLOR
{
    return tex2D(s_2D, tex);
}
```

The texture is declared with a texture variable tex0.

In this example, a sampler variable named s_2D is declared. The sampler contains the sampler state inside of curly braces. This includes the texture that will be sampled and, optionally, the filter state (that is, wrap modes, filter modes, etc.). If the sampler state is omitted, a default sampler state is applied specifying linear filtering and a wrap mode for the texture coordinates. The sampler function takes a two-component floating-point texture coordinate, and returns a two-component color. This is represented with the float2 return type and represents data in the red and green components.

Four types of samplers are defined (see [Keywords](#)) and texture lookups are performed by the intrinsic functions: [tex1D\(s, t\) \(DirectX HLSL\)](#), [tex2D\(s, t\) \(DirectX HLSL\)](#), [tex3D\(s, t\) \(DirectX HLSL\)](#), [texCUBE\(s, t\) \(DirectX HLSL\)](#). Here is an example of 3D sampling:

```
texture tex0;
sampler3D s_3D;

float3 sample_3D(float3 tex : TEXCOORD0) : COLOR
{
    return tex3D(s_3D, tex);
}
```

This sampler declaration uses default sampler state for the filter settings and address mode.

Here is the corresponding cube sampling example:

```
texture tex0;
samplerCUBE s_CUBE;

float3 sample_CUBE(float3 tex : TEXCOORD0) : COLOR
{
    return texCUBE(s_CUBE, tex);
}
```

And finally, here is the 1D sampling example:

```
texture tex0;
sampler1D s_1D;

float sample_1D(float tex : TEXCOORD0) : COLOR
{
    return tex1D(s_1D, tex);
}
```

Because the runtime does not support 1D textures, the compiler will use a 2D texture with the knowledge that the y-coordinate is unimportant. Since [tex1D\(s, t\) \(DirectX HLSL\)](#) is implemented as a 2D texture lookup, the compiler is free to choose the y-component in an efficient manner. In some rare scenarios, the compiler cannot choose an efficient y-component, in which case it will issue a warning.

```
texture tex0;
sampler s_1D_float;

float4 main(float texCoords : TEXCOORD) : COLOR
{
```

```
    return tex1D(s_1D_float, texCoords);  
}
```

This particular example is inefficient because the compiler must move the input coordinate into another register (because a 1D lookup is implemented as a 2D lookup and the texture coordinate is declared as a float1). If the code is rewritten using a float2 input instead of a float1, the compiler can use the input texture coordinate because it knows that y is initialized to something.

```
texture tex0;  
sampler s_1D_float2;  
  
float4 main(float2 texCoords : TEXCOORD) : COLOR  
{  
    return tex1D(s_1D_float2, texCoords);  
}
```

All texture lookups can be appended with "bias" or "proj" (that is, [tex2Dbias \(DirectX HLSL\)](#), [texCUBEproj \(DirectX HLSL\)](#)). With the "proj" suffix, the texture coordinate is divided by the w-component. With "bias," the mip level is shifted by the w-component. Thus, all texture lookups with a suffix always take a float4 input. [tex1D\(s, t\) \(DirectX HLSL\)](#) and [tex2D\(s, t\) \(DirectX HLSL\)](#) ignore the yz- and z-components respectively.

Samplers may also be used in array, although no back end currently supports dynamic array access of samplers. Therefore, the following is valid because it can be resolved at compile time:

```
tex2D(s[0],tex)
```

However, this example is not valid.

```
tex2D(s[a],tex)
```

Dynamic access of samplers is primarily useful for writing programs with literal loops. The following code illustrates sampler array accessing:

```
sampler sm[4];\n\nfloat4 main(float4 tex[4] : TEXCOORD) : COLOR\n{\n    float4 retColor = 1;\n\n    for(int i = 0; i < 4;i++)\n    {\n        retColor *= tex2D(sm[i],tex[i]);\n    }\n\n    return retColor;\n}
```

ⓘ Note

Using the Microsoft Direct3D debug runtime can help you catch mismatches between the number of components in a texture and a sampler.

Writing Functions

Functions break large tasks into smaller ones. Small tasks are easier to debug and can be reused, once proven. Functions can be used to hide details of other functions, which makes a program composed of functions easier to follow.

HLSL functions are similar to C functions in several ways: They both contain a definition and a function body and they both declare return types and argument lists. Like C functions, HLSL validation does type checking on the arguments, argument types, and the return value during shader compilation.

Unlike C functions, HLSL entry point functions use semantics to bind function arguments to shader inputs and outputs (HLSL functions called internally ignore semantics). This makes it easier to bind buffer data to a shader, and bind shader outputs to shader inputs.

A function contains a declaration and a body, and the declaration must precede the body.

```
float4 VertexShader_Tutorial_1(float4 inPos : POSITION ) : POSITION\n{\n
```

```
    return mul(inPos, WorldViewProj );
}
```

The function declaration includes everything in front of the curly braces:

```
float4 VertexShader_Tutorial_1(float4 inPos : POSITION ) : POSITION
```

A function declaration contains:

- A return type
- A function name
- An argument list (optional)
- An output semantic (optional)
- An annotation (optional)

The return type can be any of the HLSL basic data types such as a float4:

```
float4 VertexShader_Tutorial_1(float4 inPos : POSITION ) : POSITION
{
    ...
}
```

The return type can be a structure that has already been defined:

```
struct VS_OUTPUT
{
    float4 vPosition      : POSITION;
    float4 vDiffuse       : COLOR;
};

VS_OUTPUT VertexShader_Tutorial_1(float4 inPos : POSITION )
{
    ...
}
```

If the function does not return a value, void can be used as the return type.

```
void VertexShader_Tutorial_1(float4 inPos : POSITION )
{
```

```
    ...  
}
```

The return type always appears first in a function declaration.

```
float4 VertexShader_Tutorial_1(float4 inPos : POSITION ) : POSITION
```

An argument list declares the input arguments to a function. It may also declare values that will be returned. Some arguments are both input and output arguments. Here is an example of a shader that takes four input arguments.

```
float4 Light(float3 LightDir : TEXCOORD1,  
            uniform float4 LightColor,  
            float2 texcrd : TEXCOORD0,  
            uniform sampler samp) : COLOR  
{  
    float3 Normal = tex2D(samp,texcrd);  
  
    return dot((Normal*2 - 1), LightDir)*LightColor;  
}
```

This function returns a final color, that is a blend of a texture sample and the light color. The function takes four inputs. Two inputs have semantics: LightDir has the [TEXCOORD1](#) semantic, and texcrd has the [TEXCOORD0](#) semantic. The semantics mean that the data for these variables will come from the vertex buffer. Even though the LightDir variable has a [TEXCOORD1](#) semantic, the parameter is probably not a texture coordinate. The [TEXCOORDn](#) semantic type is often used to supply a semantic for a type that is not predefined (there is no vertex shader input semantic for a light direction).

The other two inputs LightColor and samp are labeled with the [uniform](#) keyword. These are uniform constants that will not change between draw calls. The values for these parameters come from shader global variables.

Arguments can be labeled as inputs with the [in](#) keyword, and output arguments with the [out](#) keyword. Arguments cannot be passed by reference; however, an argument can be both an input and an output if it is declared with the [inout](#) keyword. Arguments passed to a function that are marked with the [inout](#) keyword are considered copies of the original until the function returns, and they are copied back. Here's an example using [inout](#):

```
void Increment_ByVal(inout float A, inout float B)
{
    A++; B++;
}
```

This function increments the values in A and B and returns them.

The function body is all of the code after the function declaration.

```
float4 VertexShader_Tutorial_1(float4 inPos : POSITION ) : POSITION
{
    return mul(inPos, WorldViewProj );
};
```

The body consists of statements which are surrounded by curly braces. The function body implements all of the functionality using variables, literals, expressions, and statements.

The shader body does two things: it performs a matrix multiply and returns a float4 result. The matrix multiply is accomplished with the [mul \(DirectX HLSL\)](#) function, which performs a 4x4 matrix multiply. [mul \(DirectX HLSL\)](#) is called an intrinsic function because it is already built into the HLSL library of functions. Intrinsic functions will be covered in more detail in the next section.

The matrix multiply combines an input vector Pos and a composite matrix WorldViewProj. The result is position data transformed into screen space. This is the minimum vertex shader processing we can do. If we were using the fixed function pipeline instead of a vertex shader, the vertex data could be drawn after doing this transform.

The last statement in a function body is a return statement. Just like C, this statement returns control from the function to the statement that called the function.

Function return types can be any of the simple data types defined in HLSL, including bool, int half, float, and double. Return types can be one of the complex data types such as vectors and matrices. HLSL types that refer to objects cannot be used as return types. This includes pixelshader, vertexshader, texture, and sampler.

Here is an example of a function that uses a structure for a return type.

```
float4x4 WorldViewProj : WORLDVIEWPROJ;

struct VS_OUTPUT
{
    float4 Pos : POSITION;
};

VS_OUTPUT VS_HLL_Example(float4 inPos : POSITION )
{
    VS_OUTPUT Out;

    Out.Pos = mul(inPos,  WorldViewProj );

    return Out;
};
```

The `float4` return type has been replaced with the structure `VS_OUTPUT`, which now contains a single `float4` member.

A return statement signals the end of a function. This is the simplest return statement. It returns control from the function to the calling program. It returns no value.

```
void main()
{
    return ;
}
```

A return statement can return one or more values. This example returns a literal value:

```
float main( float input : COLOR0 ) : COLOR0
{
    return 0;
}
```

This example returns the scalar result of an expression:

```
return light.enabled;
```

This example returns a `float4` constructed from a local variable and a literal:

```
return float4(color.rgb, 1) ;
```

This example returns a float4 that is constructed from the result returned from an intrinsic function, and a few literal values:

```
float4 func(float2 a: POSITION): COLOR
{
    return float4(sin(length(a) * 100.0) * 0.5 + 0.5, sin(a.y * 50.0), 0,
1);
}
```

This example returns a structure that contains one or more members:

```
float4x4 WorldViewProj;

struct VS_OUTPUT
{
    float4 Pos : POSITION;
};

VS_OUTPUT VertexShader_Tutorial_1(float4 inPos : POSITION )
{
    VS_OUTPUT out;
    out.Pos = mul(inPos, WorldViewProj );
    return out;
};
```

Flow Control

Most current vertex and pixel shader hardware is designed to run a shader line by line, executing each instruction once. HLSL supports flow control, which includes static branching, predicated instructions, static looping, dynamic branching, and dynamic looping.

Previously, using an if statement resulted in assembly-language shader code that implements both the if side and the else side of the code flow. Here is an example of the in HLSL code that was compiled for vs_1_1:

```
if (Value > 0)
    oPos = Value1;
```

```
    else
        oPos = Value2;
```

And here is the resulting assembly code:

```
// Calculate linear interpolation value in r0.w
mov r1.w, c2.x
slt r0.w, c3.x, r1.w
// Linear interpolation between value1 and value2
mov r7, -c1
add r2, r7, c0
mad oPos, r0.w, r2, c1
```

Some hardware allows for either static or dynamic looping, but most require linear execution. On the models that do not support looping, all loops must be unrolled. An example is the [DepthOfField Sample](#) sample that uses unrolled loops even for ps_1_1 shaders.

HLSL now includes support for each of these types of flow control:

- static branching
- predicated instructions
- static looping
- dynamic branching
- dynamic looping

Static branching allows blocks of shader code to be switched on or off based on a Boolean shader constant. This is a convenient method for enabling or disabling code paths based on the type of object currently being rendered. Between draw calls, you can decide which features you want to support with the current shader and then set the Boolean flags required to get that behavior. Any statements that are disabled by a Boolean constant are skipped during shader execution.

The most familiar branching support is dynamic branching. With dynamic branching, the comparison condition resides in a variable, which means that the comparison is done for each vertex or each pixel at run time (as opposed to the comparison occurring at compile time, or between two draw calls). The performance hit is the cost of the branch plus the cost of the instructions on the side of the branch taken. Dynamic branching is implemented in shader model 3 or higher. Optimizing shaders that work with these models is similar to optimizing code that runs on a CPU.

Related topics

Programming Guide for HLSL

Using Shaders in Direct3D 9

Article • 08/19/2020 • 4 minutes to read

- [Compiling a Shader for Specific Hardware](#)
- [Initializing Shader Constants](#)
- [Binding a Shader Parameter to a Particular Register](#)
- [Rendering a Programmable Shader](#)
- [Debugging Shaders](#)
- [Related topics](#)

Compiling a Shader for Specific Hardware

Shaders were first added to Microsoft DirectX in DirectX 8.0. At that time, several virtual shader machines were defined, each roughly corresponding to a particular graphics processor produced by the top 3D graphics vendors. For each of these virtual shader machines, an assembly language was designed. Programs written to the shader models (names vs_1_1 and ps_1_1 - ps_1_4) were relatively short and were generally written by developers directly in the appropriate assembly language. The application would pass this human-readable assembly language code to the D3DX library using [D3DXAssembleShader](#) and get back a binary representation of the shader which would in turn get passed using [CreateVertexShader](#) or [CreatePixelShader](#). For more detail, see the software development kit (SDK).

The situation in Direct3D 9 is similar. An application passes an HLSL shader to D3DX using [D3DXCompileShader](#) and gets back a binary representation of the compiled shader which in turn is passed to Microsoft Direct3D using [CreatePixelShader](#) or [CreateVertexShader](#). The runtime does not know anything about HLSL, only the binary assembly shader models. This is nice because it means that the HLSL compiler can be updated independent of the Direct3D runtime. You can also compile shaders offline using [fxc](#).

In addition to the development of the HLSL compiler, Direct3D 9 also introduced the assembly-level shader models to expose the functionality of the latest generation of graphics hardware. Application developers can work in assembly for these new models (vs_2_0, vs_3_0, ps_2_0, ps_3_0) but we expect most developers to move to HLSL for shader development.

Of course, the ability to write an HLSL program to express a particular shading algorithm does not automatically enable it to run on any given hardware. An application calls D3DX to compile a shader into binary assembly code with [D3DXCompileShader](#). One of

the limitations with this entry point is a parameter that defines which of the assembly-level models (or compilation targets) the HLSL compiler should use to express the final shader code. If an application is doing HLSL shader compilation at run time (as opposed to compile time or offline), the application could examine the capabilities of the Direct3D device and select the compilation target to match. If the algorithm expressed in the HLSL shader is too complex to execute on the selected compilation target, compilation will fail. This means that while HLSL is a huge benefit to shader development, it does not free developers from the realities of shipping games to a target audience with graphics devices of varying capabilities. As a game developer, you still have to manage a tiered approach to your visuals; this means writing better shaders for more capable graphics cards and writing more basic versions for older cards. With well written HLSL, however, this burden can be eased significantly.

Rather than compile HLSL shaders using D3DX on the customer's machine at application load time or on first use, many developers choose to compile their shader from HLSL to binary assembly code before they even ship. This keeps their HLSL source code away from prying eyes and also ensures that all the shaders their application will ever run have gone through their internal quality assurance process. A convenient utility for compiling shaders offline is [fxc](#). This tool has a number of options that you can use to compile code for the specified compile target. Studying the disassembled output can be very educational during development if you want to optimize your shaders or just generally get to know the virtual shader machine's capabilities at a more detailed level. These options are summarized below:

Initializing Shader Constants

Shader constants are contained in the constant table. This can be accessed with the [ID3DXConstantTable](#) interface. Global shader variables can be initialized in shader code. These are initialized at run time by calling [SetDefaults](#).

Binding a Shader Parameter to a Particular Register

The compiler will automatically assign registers to global variables. The compiler would assign Environment to sampler register s0, SparkleNoise to sampler register s1, and k_s to constant register c0 (assuming no other sampler or constant registers were already assigned) for the following three global variables:

```
sampler Environment;  
sampler SparkleNoise;  
float4 k_s;
```

It is also possible to bind variables to a specific register. To force the compiler to assign to a particular register, use the following syntax:

```
register(RegisterName)
```

where RegisterName is the name of the specific register. The following examples demonstrate the specific register assignment syntax, where the sampler Environment will be bound to sampler register s1, SparkleNoise will be bound to sampler register s0, and k_s will be bound to constant register c12:

```
sampler Environment : register(s1);  
sampler SparkleNoise : register(s0);  
float4 k_s : register(c12);
```

Rendering a Programmable Shader

A shader is rendered by setting the current shader in the device, initializing the shader constants, telling the device where the varying input data is coming from, and finally rendering the primitives. Each of these can be accomplished by calling the following methods respectively:

- [SetVertexShader](#)
- [SetVertexShaderConstantF](#)
- [SetStreamSource](#)
- [DrawPrimitive](#)

Debugging Shaders

The DirectX extension for Microsoft Visual Studio .NET provides a fully integrated HLSL debugger within the Visual Studio .NET Integrated Development Environment (IDE). In order to prepare for shader debugging, you must install the right tools on your machine (see [Debugging Shaders in Visual Studio \(Direct3D 9\)](#)).

Related topics

[Programming Guide for HLSL](#)

Using Shaders in Direct3D 10

Article • 06/08/2021 • 5 minutes to read

The pipeline has three shader stages and each one is programmed with an HLSL shader. All Direct3D 10 shaders are written in HLSL, targeting shader model 4.

Differences between Direct3D 9 and Direct3D 10:

- Unlike Direct3D 9 shader models which could be authored in an intermediate assembly language, shader model 4.0 shaders are only authored in HLSL. Offline compilation of shaders into device-consumable bytecode is still supported, and recommended for most scenarios.

This example uses only a vertex shader. Because all shaders are built from the common shader core, learning how to use a vertex shader is very similar to using a geometry or pixel shader.

Once you have authored an HLSL shader (this example uses the vertex shader `HLSLWithoutFX.vsh`), you will need to prepare it for the particular pipeline stage that will use it. To do this you need to:

- [Compile a Shader](#)
- [Create a Shader Object](#)
- [Set the Shader Object](#)
- [Repeat for all 3 Shader Stages](#)

These steps need to be repeated for each shader in the pipeline.

Compile a Shader

The first step is to compile the shader, to check to see that you have coded the HLSL statements correctly. This is done by calling `D3D10CompileShader` and supplying it with several parameters as shown here:

```
IPD3D10Blob * pBlob;  
  
// Compile the vertex shader from the file  
D3D10CompileShader( strPath, strlen( strPath ), "HLSLWithoutFX.vsh",  
    NULL, NULL, "Ripple", "vs_4_0", dwShaderFlags, &pBlob, NULL );
```

This function takes the following parameters:

- The name of the file (and length of the name string in bytes) that contains the shader. This example uses a vertex shader only (in the file HLSLWithoutFX.vsh file where the file extension .vsh is an abbreviation for vertex shader).
- The shader function name. This example compiles a vertex shader from the Ripple function which takes a single input and returns an output struct (the function is from the HLSLWithoutFX sample):

```
VS_OUTPUT Ripple( in float2 vPosition : POSITION )
```

- A pointer to all macros used by the shader. Use D3D10_SHADER_MACRO to help define your macros; simply create a name string that contains all the macro names (with each name separated by a space) and a definition string (with each macro body separated by a space). Both strings need to be NULL terminated.
- A pointer to any other files that you need included to get your shaders to compile. This uses the ID3D10Include interface which has two user-implemented methods: Open and Close. To make this work, you will need to implement the body of the Open and Close methods; in the Open method add the code you would use to open whatever include files you want, in the Close function add the code to close the files when you are done with them.
- The name of the shader function to compile. This shader compiles the Ripple function.
- The shader profile to target when compiling. Since you can compile a function into a vertex, geometry, or pixel shader, the profile tells the compiler which type of shader and which shader model to compare the code against.
- Shader compiler flags. These flags tell the compiler what information to put into the compiled output and how you want the output code optimized: for speed, for debug, etc. See [Effect Constants \(Direct3D 10\)](#) for a listing of the available flags. The sample contains some code you can use to set the compiler flag values for your project - this is mainly a question of whether or not you want to generate debug information.
- A pointer to the buffer that contains the compiled shader code. The buffer also contains any embedded debug and symbol table information requested by the compiler flags.

- A pointer to a buffer that contains a listing of errors and warnings that were encountered during the compile, which are the same messages you would see in the debug output if you were running the debugger while compiling the shader. **NULL** is an acceptable value when you don't want the errors returned to a buffer.

If the shader compiles successfully, a pointer to the shader code is returned as a **ID3D10Blob** interface. It is called the Blob interface because the pointer is to a location in memory that is made up of an array of **DWORD**'s. The interface is provided so that you can get a pointer to the compiled shader which you will need in the next step.

Beginning with the December 2006 SDK, the DirectX 10 HLSL compiler is now the default compiler in both DirectX 9 and DirectX 10. See [Effect-Compiler Tool](#) for details.

Get a Pointer to a Compiled Shader

Several API methods require a pointer to a compiled shader. This argument is usually called *pShaderBytecode* because it points to a compiled shader represented as a sequence of byte codes. To get a pointer to a compiled shader, first compile the shader by calling [D3D10CompileShader](#) or a similar function. If compilation is successful, the compiled shader is returned in an **ID3D10Blob** interface. Finally, use the [GetBufferPointer](#) method to return the pointer.

Create a Shader Object

Once the shader is compiled, call [CreateVertexShader](#) to create the shader object:

```
ID3D10VertexShader ** ppVertexShader
ID3D10Blob pBlob;

// Create the vertex shader
hr = pd3dDevice->CreateVertexShader( (DWORD*)pBlob->GetBufferPointer(),
                                         pBlob->GetBufferSize(), &ppVertexShader );

// Release the pointer to the compiled shader once you are done with it
pBlob->Release();
```

To create the shader object, pass the pointer to the compiled shader into [CreateVertexShader](#). Since you had to successfully compile the shader first, this call will almost certainly pass, unless you have a memory problem on your machine.

You can create as many shader objects as you like and simply keep pointers to them. This same mechanism works for geometry and pixel shaders assuming you match the shader profiles (when you call the compile method) to the interface names (when you call the create method).

Set the Shader Object

The last step is set the shader to the pipeline stage. Since there are three shader stages in the pipeline, you will need to make three API calls, one for each stage.

```
// Set a vertex shader  
pd3dDevice->VSSetShader( g_pVS10 );
```

The call to VSSetShader takes the pointer to the vertex shader created in step 1. This sets the shader in the device. The vertex shader stage is now initialized with its vertex shader code, all that remains is initializing any shader variables.

Repeat for all 3 Shader Stages

Repeat these same set of steps to build any vertex or pixel shader or even a geometry shader that outputs to the pixel shader.

Related Topics

[Compiling Shaders](#)

Related topics

[Programming Guide for HLSL](#)

Optimizing HLSL Shaders

Article • 08/23/2019 • 2 minutes to read

This section describes general-purpose strategies that you can use to optimize your shaders. You can apply these strategies to shaders that are written in any language, on any platform.

- [Know Where To Perform Shader Calculations](#)
- [Skip Unnecessary Instructions](#)
- [Pack Variables and Interpolants](#)
- [Reduce Shader Complexity](#)
- [Related Topics](#)
- [Related topics](#)

Know Where To Perform Shader Calculations

Vertex shaders perform operations that include fetching vertices and performing matrix transformation of vertex data. Typically, vertex shaders are executed once per vertex.

Pixel Shaders perform operations that include fetching texture data and performing lighting calculations. Typically, pixel shaders are executed once per pixel for a given piece of geometry.

Typically, pixels outnumber vertices in a scene, so pixel shaders execute more often than vertex shaders.

When you design shader algorithms, keep the following in mind:

- Perform calculations on the vertex shader if possible. A calculation that is performed on a pixel shader is much more expensive than a calculation that is performed on a vertex shader.
- Consider using per-vertex calculations to improve performance in situations such as dense meshes. For dense meshes, per-vertex calculations may produce results that are visually indistinguishable from results produced with per-pixel calculations.

Skip Unnecessary Instructions

In HLSL, dynamic branching provides the ability to limit the number of instructions that are executed. Therefore, dynamic branching can help speed up shader execution time. If geometry or pixels are not displayed, use dynamic branching to exit the shader or to

limit instructions. For example, if a pixel is not lit, there is no point in executing the lighting algorithm.

The following table lists some cases where you can test conditions in your shader and use dynamic branching to skip unnecessary instructions. The table is not comprehensive. Rather, it is intended to give you ideas for optimizing your code.

Condition to Check	Response in the Shader
Alpha check determines that a pixel will not be seen.	Skip the rest of the shader.
The pixel or geometry is fully fogged.	Skip the rest of the shader.
Skin weights are zero.	Skip bones.
Light attenuation is zero.	Skip lighting.
Non-positive Lambertian term.	Skip lighting.

Pack Variables and Interpolants

Be mindful of the space required for shader data. Pack as much information into a variable or interpolant as possible. Sometimes, the information from two variables can be packed into the memory space of a single variable.

Reduce Shader Complexity

Keep your shaders small and simple. In general, shaders with fewer instructions execute more quickly than shaders with more instructions. It is also easier to debug and optimize smaller, less complex shaders.

Related Topics

[Programming Guide for HLSL](#)

Related topics

[Programming Guide for HLSL](#)

Debugging Shaders in Visual Studio

Article • 05/31/2022 • 2 minutes to read

The latest tool for debugging shaders now ships as a feature in Microsoft Visual Studio, called Visual Studio Graphics Debugger. This new tool is a replacement for the [PIX](#) for Windows tool. Visual Studio Graphics Debugger has greatly improved usability, support for Windows 8 and Direct3D 11.1, and integration with traditional Visual Studio features such as call stacks and debugging windows for [HLSL](#) debugging. For more info about this new feature, see [Debugging DirectX Graphics](#).

Related topics

[Programming Guide for HLSL](#)

Compiling shaders

Article • 05/16/2022 • 3 minutes to read

ⓘ Note

This topic covers the `FXC.EXE` compiler used for Shader Models 2 through 5.1. For Shader Model 6, you use `DXC.EXE` instead, which is documented in [Using dxc.exe and dxcompiler.dll](#). Visual Studio will use `DXC.EXE` automatically when Shader Model 6 is selected for the HLSL Property Page configuration.

Microsoft Visual Studio can compile shader code from `*.hlsl` and `*.fx` files that you include in your C++ project.

As part of the build process, Visual Studio uses the `fxc.exe` or [dxc.exe](#) HLSL code compiler to compile the HLSL shader source files into binary shader object files or into byte arrays that are defined in header files. How the HLSL code compiler compiles each shader source file in your project depends on how you specify the **Output Files** property for that file. For more info about HLSL property pages, see [HLSL Property Pages](#).

The compile method that you use typically depends on the size of your HLSL shader source file. If you include a large amount of byte code in a header, you increase the size and the initial load time of your app. You also force all byte code to reside in memory even after the shader is created, which wastes resources. But when you include byte code in a header, you can reduce code complexity and simplify shader creation.

Let's now look at various ways to compile your shader code and conventions for file extensions for shader code.

- [Using shader code file extensions](#)
- [Compiling at build time to object files](#)
- [Compiling at build time to header files](#)
- [Compiling with D3DCompileFromFile](#)
- [Related Topics](#)
- [Related topics](#)

Using shader code file extensions

To conform to Microsoft convention, use these file extensions for your shader code:

- A file with the .hlsl extension holds High Level Shading Language ([HLSL](#)) source code. The older .fx extension is also supported, but is usually associated with the [legacy Effects system](#).
- A file with the .cso extension holds a compiled shader object.
- A file with the .h extension is a header file, but in a shader code context, this header file defines a byte array that holds shader data. Other common extensions for HLSL shader code headers include .hlsl and .fxh.

Compiling at build time to object files

If you compile your .hlsl files into binary shader object files, your app needs to read the data from those object files (.cso is the default extension for these object files), assign the data to byte arrays, and create shader objects from those byte arrays. For example, to create a vertex shader ([ID3D11VertexShader](#)^{**}), call the [ID3D11Device::CreateVertexShader](#) method with a byte array that contains compiled vertex shader byte code. In this example code, the **Ouput Files** property for the SimpleVertexShader.hlsl file specifies to compile into the SimpleVertexShader.cso object file.

C++

```
auto vertexShaderBytecode = ReadData("SimpleVertexShader.cso");
ComPtr<ID3D11VertexShader> vertexShader;
DX::ThrowIfFailed(
    m_d3dDevice->CreateVertexShader(
        vertexShaderBytecode->Data,
        vertexShaderBytecode->Length,
        nullptr,
        &vertexShader
    )
)
```

The `ReadData` helper here will look in the current working directory as well as the same directory as the current process' EXE file as the .cso files are typically found alongside other VS build products. See [ReadData.h](#) for an example implementation.

Compiling at build time to header files

If you compile your .hlsl files into byte arrays that are defined in header files, you need to include those header files in your code. In this example code, the **Ouput Files** property for the PixelShader.hlsl file specifies to compile into the `g_psshader` byte array that is defined in the PixelShader.h header file.

C++

```
namespace
{
    include "PixelShader.h"
}
...
ComPtr<ID3D11PixelShader> m_pPixelShader;
hr = pDevice->CreatePixelShader(g_psshader, sizeof(g_psshader),
nullptr, &m_pPixelShader);
```

Compiling with D3DCompileFromFile

You can also use the [D3DCompileFromFile](#) function at run time to compile shader code for Direct3D 11. For more info about how to do this, see [How To: Compile a Shader](#).

ⓘ Note

Windows Store apps support using [D3DCompileFromFile](#) for development but not for deployment.

Related Topics

[Programming Guide for HLSL](#)

Related topics

[Programming Guide for HLSL](#)

Specifying Compiler Targets

Article • 08/19/2021 • 3 minutes to read

You need to specify the shader target — set of shader features — to compile against when you call the [D3DCompile](#), [D3DCompile2](#), or [D3DCompileFromFile](#) function. Here we list the targets for various profiles that the [D3DCompile*](#) functions and the HLSL compiler support.

- [Direct3D 11.0 and 11.1 feature levels](#)
- [Direct3D 10.1 feature level](#)
- [Direct3D 10.0 feature level](#)
- [Direct3D 9.1, 9.2, and 9.3 feature levels](#)
- [Legacy Direct3D 9 Shader Model 3.0](#)
- [Legacy Direct3D 9 Shader Model 2.0](#)
- [Legacy Direct3D 9 Shader Model 1.x](#)
- [Legacy Effects](#)
- [Notes](#)
- [Related topics](#)

Direct3D 11.0 and 11.1 feature levels

Here are the shader targets that Direct3D 11.0 and 11.1 [feature levels](#) support.

Target	Description
cs_5_0	DirectCompute 5.0 (compute shader)
ds_5_0	Domain shader
gs_5_0	Geometry shader ↗
hs_5_0	Hull shader
ps_5_0	Pixel shader ↗
vs_5_0	Vertex shader ↗

Direct3D 10.1 feature level

Here are the shader targets that the Direct3D 10.1 [feature level](#) supports.

Target	Description
cs_4_1	DirectCompute 4.1 (compute shader) ¹
gs_4_1	Geometry shader
ps_4_1	Pixel shader
vs_4_1	Vertex shader

Direct3D 10.0 feature level

Here are the shader targets that the Direct3D 10.0 [feature level](#) supports.

Target	Description
cs_4_0	DirectCompute 4.0 (compute shader) ¹
gs_4_0	Geometry shader
ps_4_0	Pixel shader
vs_4_0	Vertex shader

Direct3D 9.1, 9.2, and 9.3 feature levels

Here are the shader targets that Direct3D 9.1, 9.2 and 9.3 [feature levels](#) support.

ⓘ Note

When you use the *_4_0_level_9_x HLSL shader profiles, you implicitly use of the **Shader Model 2.x** profiles to support Direct3D 9 capable hardware. Shader Model 2.x profiles support more limited flow control behavior than the **Shader Model 4.x** and later profiles.

Target	Description

Target	Description
ps_4_0_level_9_1	<p>Pixel shader for 9.1 and 9.2 (similar limits to ps_2_0)</p> <ul style="list-style-type: none"> • 64 arithmetic and 32 texture instructions • 12 temporary registers • 4 levels of dependent reads
ps_4_0_level_9_3	<p>Pixel shader for 9.3 (similar limits to ps_2_x² with additional shader features)</p> <ul style="list-style-type: none"> • 512 instructions • 32 temporary registers • Static flow control (max depth of 4) • Dynamic flow control (max depth of 24) • D3DPS20CAPS_ARBITRARYSWIZZLE • D3DPS20CAPS_GRADIENTINSTRUCTIONS • D3DPS20CAPS_PREDICATION • D3DPS20CAPS_NODEPENDENTREADLIMIT • D3DPS20CAPS_NOTEXINSTRUCTIONLIMIT
vs_4_0_level_9_1	<p>Vertex shader for 9.1 and 9.2 (similar to vs_2_0)</p> <ul style="list-style-type: none"> • 256 instructions • 12 temporary registers • Static flow control (max depth of 1)
vs_4_0_level_9_3	<p>Vertex shader for 9.3 (similar to vs_2_a² with additional shader features and instancing)</p> <ul style="list-style-type: none"> • 256 instructions • 32 temporary registers • Static flow control (max depth of 4) • D3DVS20CAPS_PREDICATION

Legacy Direct3D 9 Shader Model 3.0

Here are the shader targets for legacy Direct3D 9 shader model 3.0³.

Target	Description
ps_3_0	Pixel shader 3.0
ps_3_sw	Pixel shader 3.0 (software)

Target	Description
vs_3_0	Vertex shader 3.0
vs_3_sw	Vertex shader 3.0 (software)

Legacy Direct3D 9 Shader Model 2.0

Here are the shader targets for legacy Direct3D 9 shader model 2.0³.

Target	Description
ps_2_0	Pixel shader 2.0
ps_2_a	Pixel shader 2a
ps_2_b	Pixel shader 2b
ps_2_sw	Pixel shader 2.0 software
vs_2_0	Vertex shader 2.0
vs_2_a	Vertex shader 2a
vs_2_sw	Vertex shader 2.0 software

Legacy Direct3D 9 Shader Model 1.x

Here are the shader targets for legacy Direct3D 9 shader model 1.x⁴.

Target	Description
tx_1_0	Texture shader profile that legacy D3DX9 ⁵ functions D3DXCreateTextureShader and D3DXFillTextureTX use
vs_1_1	Vertex shader 1.1

Legacy Effects

Here are the effect targets for legacy effects.

Target	Description
fx_2_0	Effects (FX) for Direct3D 9 in D3DX9 ⁵
fx_4_0	Effects (FX) for Direct3D 10.0 in D3DX10 ⁵
fx_4_1	Effects (FX) for Direct3D 10.1 in D3DX10 ⁵
fx_5_0	Effects (FX) for Direct3D 11 ⁵

Notes

Here are some notes that the preceding sections refer to:

1. **feature level** 10.0 and 10.1 devices can optionally support DirectCompute. To verify support, use [ID3D11Device::CheckFeatureSupport](#) with [D3D11_FEATURE_D3D10_X_HARDWARE_OPTIONS](#).
2. **feature level** 9.3 effectively requires hardware that complies with the requirements for [legacy Direct3D 9 shader model 3.0](#), but this feature level does not make use of vs_3_0 or ps_3_0 targets.
3. Only use legacy Direct3D 9 shader models with the Direct3D 9 API. Instead, use the 9.x profiles with the Direct3D 10.x and 11.x API.
4. The current HLSL shader [D3DCompile*](#) functions don't support legacy 1.x pixel shaders. The last version of HLSL to support these targets was D3DX9 in the October 2006 release of the DirectX SDK.
5. All versions of D3DX and the DirectX SDK are deprecated. For more info, see [Where is the DirectX SDK?](#).

Related topics

[Programming Guide for HLSL](#)

Unpacking and Packing DXGI_FORMAT for In-Place Image Editing

Article • 04/02/2021 • 5 minutes to read

The D3DX_DXGIFormatConvert.inl file contains inline format conversion functions that you can use in the compute shader or pixel shader on Direct3D 11 hardware. You can use these functions in your application to simultaneously both read from and write to a texture. That is, you can perform in-place image editing. To use these inline format conversion functions, include the D3DX_DXGIFormatConvert.inl file in your application.

The D3DX_DXGIFormatConvert.inl header ships in the legacy DirectX SDK. It is also included in the [Microsoft.DXSDK.D3DX](#) NuGet package.

Direct3D 11's Unordered Access View (UAV) of a Texture1D, Texture2D, or Texture3D resource supports random access reads and writes to memory from a compute shader or pixel shader. However, Direct3D 11 supports simultaneously both reading from and writing to only the DXGI_FORMAT_R32_UINT texture format. For example, Direct3D 11 does not support simultaneously both reading from and writing to other, more useful formats, such as DXGI_FORMAT_R8G8B8A8_UNORM. You can use only a UAV to random access write to such other formats, or you can use only a Shader Resource View (SRV) to random access read from such other formats. Format conversion hardware is not available to simultaneously both read from and write to such other formats.

However, you can still simultaneously both read from and write to such other formats by casting the texture to the DXGI_FORMAT_R32_UINT texture format when you create a UAV, as long as the original format of the resource supports casting to DXGI_FORMAT_R32_UINT. Most 32 bit per element formats support casting to DXGI_FORMAT_R32_UINT. By casting the texture to the DXGI_FORMAT_R32_UINT texture format when you create a UAV, you can then simultaneously perform reads and writes to the texture as long as the shader performs manual format unpacking on read and packing on write.

The benefit of casting the texture to the DXGI_FORMAT_R32_UINT texture format is that later on you can use the appropriate format (for example, DXGI_FORMAT_R16G16_FLOAT) with other views on the same texture, such as Render Target Views (RTVs) or SRVs. Therefore, the hardware can perform the typical automatic format unpack and pack, can perform texture filtering, and so on where there are no hardware limitations.

The following scenario requires that an application take the following sequence of actions to perform in-place image editing.

Suppose you want to make a texture on which you can use a pixel shader or compute shader to perform in-place editing, and you want the texture data to be stored in a format that is a descendant of one of the following TYPELESS formats:

- DXGI_FORMAT_R10G10B10A2_TYPELESS
- DXGI_FORMAT_R8G8B8A8_TYPELESS
- DXGI_FORMAT_B8G8R8A8_TYPELESS
- DXGI_FORMAT_B8G8R8X8_TYPELESS
- DXGI_FORMAT_R16G16_TYPELESS

For example, the DXGI_FORMAT_R10G10B10A2_UNORM format is a descendant of the DXGI_FORMAT_R10G10B10A2_TYPELESS format. Therefore, DXGI_FORMAT_R10G10B10A2_UNORM supports the usage pattern that is described in the following sequence. Formats that descend from DXGI_FORMAT_R32_TYPELESS, such as DXGI_FORMAT_R32_FLOAT, are trivially supported without requiring any of the format conversion help that is described in the following sequence.

To perform in-place image editing

1. Create a texture with the appropriate TYPELESS-dependent format that is specified in the previous scenario together with the required bind flags, such as D3D11_BIND_UNORDERED_ACCESS | D3D11_BIND_SHADER_RESOURCE.
2. For in-place image editing, create a UAV with the DXGI_FORMAT_R32_UINT format. The Direct3D 11 API typically does not allow casting between different format "families." However, the Direct3D 11 API makes an exception with the DXGI_FORMAT_R32_UINT format.
3. In the compute shader or pixel shader, use the appropriate inline format pack and unpack functions that are provided in the D3DX_DXGIFormatConvert.inl file. For example, suppose the DXGI_FORMAT_R32_UINT UAV of the texture really holds DXGI_FORMAT_R10G10B10A2_UNORM-formatted data. After the application reads a uint from the UAV into the shader, it must call the following function to unpack the texture format:

```
XMFLOAT4 D3DX_R10G10B10A2_UNORM_to_FLOAT4(UINT packedInput)
```

Then, to write to the UAV in the same shader, the application calls the following function to pack shader data into a uint that the application can write to the UAV:

```
UINT D3DX_FLOAT4_to_R10G10B10A2_UNORM(hlsl_precise XMFLOAT4  
unpackedInput)
```

4. The application can then create other views, such as SRVs, with the required format. For example, the application can create a SRV with the DXGI_FORMAT_R10G10B10A2_UNORM format if the resource was created as DXGI_FORMAT_R10G10B10A2_TYPELESS. When a shader accesses that SRV, the hardware can perform automatic type conversion as usual.

 **Note**

If the shader must write only to a UAV, or read as an SRV, none of this conversion work is needed because you can use fully typed UAVs or SRVs. The format conversion functions provided in D3DX_DXGIFormatConvert.inl are potentially useful only if you want to perform simultaneous reading from and writing to a UAV of a texture.

The following is the list of format conversion functions that are included in the D3DX_DXGIFormatConvert.inl file. These functions are categorized by the DXGI_FORMAT that they unpack and pack. Each of the supported formats descends from one of the TYPELESS formats listed in the preceding scenario and supports casting to DXGI_FORMAT_R32_UINT as a UAV.

DXGI_FORMAT_R10G10B10A2_UNORM

```
XMFLOAT4 D3DX_R10G10B10A2_UNORM_to_FLOAT4(UINT packedInput)  
UINT D3DX_FLOAT4_to_R10G10B10A2_UNORM(hlsl_precise XMFLOAT4  
unpackedInput)
```

DXGI_FORMAT_R10G10B10A2_UINT

```
XMUINT4 D3DX_R10G10B10A2_UINT_to_UINT4(UINT packedInput)  
UINT D3DX_UINT4_to_R10G10B10A2_UINT(XMUINT4 unpackedInput)
```

DXGI_FORMAT_R8G8B8A8_UNORM

```
XMFLOAT4 D3DX_R8G8B8A8_UNORM_to_FLOAT4(UINT packedInput)
UINT      D3DX_FLOAT4_to_R8G8B8A8_UNORM(hlsl_precise XMFLOAT4 unpackedInput)
```

DXGI_FORMAT_R8G8B8A8_UNORM_SRGB

```
XMFLOAT4 D3DX_R8G8B8A8_UNORM_SRGB_to_FLOAT4_inexact(UINT packedInput) *
XMFLOAT4 D3DX_R8G8B8A8_UNORM_SRGB_to_FLOAT4(UINT packedInput)
UINT      D3DX_FLOAT4_to_R8G8B8A8_UNORM_SRGB(hlsl_precise XMFLOAT4
unpackedInput)
```

ⓘ Note

The _inexact-type function uses shader instructions that do not have high enough precision to give the exact answer. The alternative function uses a lookup table stored in the shader to give an exact SRGB->float conversion.

DXGI_FORMAT_R8G8B8A8_UINT

```
XMUINT4 D3DX_R8G8B8A8_UINT_to_UINT4(UINT packedInput)
XMUINT  D3DX_UINT4_to_R8G8B8A8_UINT(XMUINT4 unpackedInput)
```

DXGI_FORMAT_R8G8B8A8_SNORM

```
XMFLOAT4 D3DX_R8G8B8A8_SNORM_to_FLOAT4(UINT packedInput)
UINT      D3DX_FLOAT4_to_R8G8B8A8_SNORM(hlsl_precise XMFLOAT4 unpackedInput)
```

DXGI_FORMAT_R8G8B8A8_SINT

```
XMINT4 D3DX_R8G8B8A8_SINT_to_INT4(UINT packedInput)
UINT    D3DX_INT4_to_R8G8B8A8_SINT(XMINT4 unpackedInput)
```

DXGI_FORMAT_B8G8R8A8_UNORM

```
XMFLOAT4 D3DX_B8G8R8A8_UNORM_to_FLOAT4(UINT packedInput)
UINT      D3DX_FLOAT4_to_B8G8R8A8_UNORM(hlsl_precise XMFLOAT4 unpackedInput)
```

DXGI_FORMAT_B8G8R8A8_UNORM_SRGB

```
XMFLOAT4 D3DX_B8G8R8A8_UNORM_SRGB_to_FLOAT4_inexact(UINT packedInput) *
XMFLOAT4 D3DX_B8G8R8A8_UNORM_SRGB_to_FLOAT4(UINT packedInput)
UINT      D3DX_FLOAT4_to_R8G8B8A8_UNORM_SRGB(hlsl_precise XMFLOAT4
unpackedInput)
```

ⓘ Note

The _inexact-type function uses shader instructions that do not have high enough precision to give the exact answer. The alternative function uses a lookup table stored in the shader to give an exact SRGB->float conversion.

DXGI_FORMAT_B8G8R8X8_UNORM

```
XMFLOAT3 D3DX_B8G8R8X8_UNORM_to_FLOAT3(UINT packedInput)
UINT      D3DX_FLOAT3_to_B8G8R8X8_UNORM(hlsl_precise XMFLOAT3 unpackedInput)
```

DXGI_FORMAT_B8G8R8X8_UNORM_SRGB

```
XMFLOAT3 D3DX_B8G8R8X8_UNORM_SRGB_to_FLOAT3_inexact(UINT packedInput) *
XMFLOAT3 D3DX_B8G8R8X8_UNORM_SRGB_to_FLOAT3(UINT packedInput)
UINT      D3DX_FLOAT3_to_B8G8R8X8_UNORM_SRGB(hlsl_precise XMFLOAT3
unpackedInput)
```

ⓘ Note

The _inexact-type function uses shader instructions that do not have high enough precision to give the exact answer. The alternative function uses a lookup table stored in the shader to give an exact SRGB->float conversion.

DXGI_FORMAT_R16G16_FLOAT

```
XMFLOAT2 D3DX_R16G16_FLOAT_to_FLOAT2(UINT packedInput)
UINT      D3DX_FLOAT2_to_R16G16_FLOAT(hlsl_precise XMFLOAT2 unpackedInput)
```

DXGI_FORMAT_R16G16_UNORM

```
XMFLOAT2 D3DX_R16G16_UNORM_to_FLOAT2(UINT packedInput)
UINT      D3DX_FLOAT2_to_R16G16_UNORM(hlsl_precise FLOAT2 unpackedInput)
```

DXGI_FORMAT_R16G16_UINT

```
XMUINT2 D3DX_R16G16_UINT_to_UINT2(UINT packedInput)
UINT      D3DX_UINT2_to_R16G16_UINT(XMUINT2 unpackedInput)
```

DXGI_FORMAT_R16G16_SNORM

```
XMFLOAT2 D3DX_R16G16_SNORM_to_FLOAT2(UINT packedInput)
UINT      D3DX_FLOAT2_to_R16G16_SNORM(hlsl_precise XMFLOAT2 unpackedInput)
```

DXGI_FORMAT_R16G16_SINT

```
XMINIT2 D3DX_R16G16_SINT_to_INT2(UINT packedInput)
UINT      D3DX_INT2_to_R16G16_SINT(XMINIT2 unpackedInput)
```

Related Topics

[Programming Guide for HLSL](#)

Related topics

Programming Guide for HLSL

Using HLSL minimum precision

Article • 08/19/2020 • 2 minutes to read

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

You can query for the minimum precision support that the graphics driver provides by calling [ID3D11Device::CheckFeatureSupport](#) with the [D3D11_FEATURE_SHADER_MIN_PRECISION_SUPPORT](#) value. For more info, see [HLSL minimum precision support](#).

- [Declare variables with minimum precision data types](#)
- [Testing your minimum precision shader code](#)
- [Related topics](#)

Declare variables with minimum precision data types

To use minimum precision in HLSL shader code, declare individual variables with types like **min16float** (**min16float4** for a vector), **min16int**, **min10float**, and so on. With these variables, your shader code indicates that it doesn't require more precision than what the variables indicate. But hardware can ignore the minimum precision indicators and run at full 32-bit precision. When your shader code is used on hardware that takes advantage of minimum precision, you use less memory bandwidth and as a result you also use less system power as long as your shader code doesn't expect more precision than it specified.

You don't need to author multiple shaders that do and don't use minimum precision. Instead, create shaders with minimum precision, and the minimum precision variables behave at full 32-bit precision if the graphics driver reports that it doesn't support any minimum precision. HLSL minimum precision shaders don't work on operating systems earlier than Windows 8 so if you plan to target earlier operating systems, you'll need to author multiple shaders, some that do and others that don't use minimum precision.

Note

Don't make data switches between different precision levels within a shader because these types of conversions are wasteful and reduce performance. The exception is that shader constants are still always 32 bit, but vendors can design graphics hardware that can freely down-convert to whatever lower precision the HLSL instruction reading might use.

By using minimum precision, you can control the precision of computations in various parts of your shader code.

The rules for HLSL minimum precision are similar to C/C++, where the types in an expression determine the precision of the operation, not the type being eventually written to.

Testing your minimum precision shader code

The reference rasterizer ([D3D_DRIVER_TYPE_REFERENCE](#)) gives you a rough idea of how minimum precision in your HLSL shader code behaves by quantizing each HLSL instruction to the specified precision. This helps you discover code that might accidentally rely on more than the minimum precision. The reference rasterizer doesn't run any faster when your HLSL shader code uses minimum precision, but you can use it to verify the correctness of your code. [WARP \(D3D_DRIVER_TYPE_WARP\)](#) doesn't support using minimum precision in HLSL shader code; WARP just runs at full 32-bit precision.

Related topics

[Programming Guide for HLSL](#)

User clip planes on feature level 9 hardware

Article • 08/19/2020 • 6 minutes to read

Starting with Windows 8, Microsoft High Level Shader Language (HLSL) supports a syntax that you can use with the Microsoft Direct3D 11 API to specify user clip planes on [feature level 9_x](#) and higher. You can use this clip-planes syntax to write a shader, and then use that shader object with the Direct3D 11 API to run on all Direct3D feature levels.

- [Background](#)
- [Syntax](#)
- [Creating clip planes in clip space on feature level 9 and higher](#)
 - [Background reading](#)
 - [10Level9 feature levels](#)
 - [Clip plane math](#)
 - [Clipping in view space](#)
 - [Projection matrix](#)
 - [Clip space clip plane](#)
- [Related topics](#)

Background

You can access user clip planes in the Microsoft Direct3D 9 API via [IDirect3DDevice9::SetClipPlane](#) and [IDirect3DDevice9::GetClipPlane](#) methods. In Microsoft Direct3D 10 and later, you can access user clip planes through the [SV_ClipDistance](#) semantic. But before Windows 8, [SV_ClipDistance](#) was not available for [feature level 9_x](#) hardware in the Direct3D 10 or Direct3D 11 APIs. So, before Windows 8, the only way to access user clip planes with feature level 9_x hardware was through the Direct3D 9 API. Direct3D Windows Store apps can't use the Direct3D 9 API. Here we describe the syntax that you can use to access user clip planes through the Direct3D 11 API on feature level 9_x and higher.

Apps use clip planes to define a set of invisible planes within the 3D world that clip (throw away) all drawn primitives. Windows won't draw any pixel that is on the negative side of any clip planes. Apps can then use clip planes to render planar reflections.

Syntax

Use this syntax to declare clip planes as function attributes in a [function declaration](#). For example, here we use the syntax on a vertex shader fragment:

syntax

```
cbuffer ClipPlaneConstantBuffer
{
    float4 clipPlane1;
    float4 clipPlane2;
};

[clipplanes(clipPlane1,clipPlane2)]
VertexShaderOutput main(VertexShaderInput input)
{
    // the rest of the vertex shader doesn't refer to the clip plane

    ...

    return output;
}
```

This example for a vertex shader fragment denotes two clip planes. It shows that you need to place the new **clipplanes** attribute within square brackets immediately before the return value of the vertex shader. Within parentheses after the **clipplanes** attribute, you provide a list of up to 6 **float4** constants that define the plane coefficients for each active clip plane. The example also shows that you need to make the coefficients of each plane reside in a constant buffer.

ⓘ Note

There is no syntax available to disable a clip plane dynamically. You must either recompile an otherwise identical shader with no **clipplanes** attribute, or your app can set the coefficients in your constant buffer to zero so that the plane no longer affects any geometry.

This syntax is available for any 4.0 or later vertex shader target, which includes `vs_4_0_level_9_1` and `vs_4_0_level_9_3`.

Creating clip planes in clip space on feature level 9 and higher

Here we show how to create clip planes in clip space on [feature level](#) 9_x and higher.

Background reading

"Introduction to 3D Game Programming with DirectX 10" by Frank D. Luna explains the graphics math background (chapters 1, 2 and 3) you need, and the various spaces and space transformations that occur in the vertex shader (sections 5.6 and 5.8).

10Level9 feature levels

In Direct3D 10 and later, you can clip in any space that makes sense, often in world space or view space. But Direct3D 9 uses clip space, which is pre perspective divide projection space. Vectors are in clip space when the vertex shader passes them to stages that follow in the [graphics pipeline](#).

When you write a Windows Store app, you must use 10Level9 feature levels ([feature level 9_x](#)) so the app can run on feature level 9_x and higher hardware. Because your app supports feature level 9_x and higher, you must also use the common capability of applying clip planes in clip space.

When you compile a vertex shader with `vs_4_0_level_9_1` or later, that vertex shader can use the `clipplanes` attribute. A Direct3D 10 or later object has a dot product of the emitted vertex that contains each of the `float4` global constants specified in the attribute. The Direct3D 9 object has enough meta data to cause the 10Level9 runtime to issue the appropriate calls to [`IDirect3DDevice9::SetClipPlane`](#).

Clip plane math

A clip plane is defined by a vector with 4 components. The first three components define an x, y, z vector that emanates from the origin in the space we want to clip. This vector implies a plane, perpendicular to the vector and running through the origin. Windows keeps all pixels on the vector side of the plane and clips all pixels behind the plane. The forth w component pushes the plane back and causes Windows to clip less (a negative w causes Windows to clip more) along the vector line. If the x, y, z components compose a unit (normalized) vector, w pushes the plane w units back.

The math that the graphics processing unit (GPU) performs to determine clipping is a simple dot product between the vertex vector (x, y, z, 1) and the clipping plane vector. This math operation creates a projection length on the clip plane vector. A negative dot product shows the vertex to be on the clipped side of the plane.

Clipping in view space

Here is our vertex in view space:

$$\mathbf{v} = \begin{bmatrix} v_x & v_y & v_z & 1 \end{bmatrix}$$

Here is our clip plane in view space:

$$\mathbf{C} = \begin{bmatrix} C_x & C_y & C_z & C_w \end{bmatrix}$$

Here is the dot product of vertex and clip plane in view space:

$$\text{ClipDistance} = \mathbf{v} \cdot \mathbf{C} = v_x C_x + v_y C_y + v_z C_z + C_w$$

This math operation works for a Direct3D 10 or later object but won't work for a Direct3D 9 object. For Direct3D 9, we must first get through our projection transform into clip space.

Projection matrix

A projection matrix transforms a vertex from view space (where the origin is the viewer's eye, $+x$ is to the right, $+y$ is up, and $+z$ is straight ahead) into clip space. The projection matrix readies the vertex for hardware clipping and the [rasterization stage](#). Here is a standard perspective matrix (other projections require different math):

r ratio of window width/height * α * viewing angle *f* distance from the viewer to the far plane *n* distance from the viewer to the near plane

![projection matrix](images/projection-matrix.png)

The next matrix is a simplified version of the previous matrix. We show the matrix simplified so we can use it later in the matrix multiply operation.

$$\mathbf{P} = \begin{bmatrix} P_x & 0 & 0 & 0 \\ 0 & P_y & 0 & 0 \\ 0 & 0 & A & 1 \\ 0 & 0 & B & 0 \end{bmatrix}$$

Now we transform our view space vertex into clip space with a matrix multiply:

$$\mathbf{v} \mathbf{P} = \begin{bmatrix} v_x P_x & v_y P_y & v_z A_y + B & v_z \end{bmatrix}$$

In our matrix multiply operation, our x and y components are only slightly adjusted, but our z and w components are quite mangled. Our clip plane won't give us what we want any more.

Clip space clip plane

Here we want to create a clip space clip plane whose dot product with our clip space vertex gives us the same value as $\mathbf{v} \cdot \mathbf{C}$ in the [Clipping in view space](#) section.

$$\mathbf{C}_P = \begin{bmatrix} C_{P_x} & C_{P_y} & C_{P_z} & C_{P_w} \end{bmatrix}$$

$$\mathbf{v} \cdot \mathbf{C} = \mathbf{v} \mathbf{P} \cdot \mathbf{C}_P$$

$$v_x C_x + v_y C_y + v_z C_z + C_w = v_x P_x C_{P_x} + v_y P_y C_{P_y} + v_z A_y C_{P_z} + B C_{P_z} + v_z C_{P_w}$$

Now we can break the preceding math operation up by vertex component into four separate equations:

$$v_x C_x = v_x P_x C_{P_x} \rightarrow C_{P_x} = \frac{C_x}{P_x}$$

$$v_y C_y = v_y P_y C_{p_y} \rightarrow C_{p_y} = \frac{C_y}{P_y}$$

$$C_w = B C_{p_z} \rightarrow C_{p_z} = \frac{C_w}{B}$$

$$v_z C_z = v_z A_y C_{p_z} + v_z C_{p_w} \rightarrow C_z = A_y C_{p_z} + C_{p_w} \rightarrow C_{p_w} = C_z - A_y C_{p_z} \rightarrow C_{p_w} = C_z - \frac{C_w A}{B}$$

Our view space clip plane and our projection matrix derive and give us our clip space clip plane.

$$\mathbf{C}_P = \begin{bmatrix} \frac{C_x}{P_x} & \frac{C_y}{P_y} & \frac{C_w}{B} & C_z - \frac{C_w A}{B} \end{bmatrix}$$

Related topics

[Programming Guide for HLSL](#)

[Function Declaration Syntax](#)

HLSL Shader Model 5

Article • 08/19/2020 • 2 minutes to read

This section contains overview material for the High-Level Shader Language, specifically the new features in shader model 5 introduced in Microsoft Direct3D 11.

In This Section

Item	Description
Dynamic Linking	Dynamic linking allows the runtime to make a decision at draw-time (rather than compile-time) about which code path to run. This reduces the shader proliferation problem caused by shaders with nearly identical input signatures.
Geometry Shader Features	New geometry shader features including: instancing, which provides a performance boost when the order of primitives in the stream doesn't matter, and multiple point output streams so a shader can output vertices on more than one stream.
Tessellation	The Direct3D 11 runtime supports three new stages that implement tessellation, which converts low-detail subdivision surfaces into higher-detail primitives on the GPU. Tessellation tiles (or breaks up) high-order surfaces into suitable structures for rendering. The three tessellation stages are hull-shader, tessellator, and domain-shader stages.

In addition, the reference section covers many new API elements for shader model 5 including: [attributes](#), [intrinsic functions](#), [shader model 5 objects and methods](#), and [system values](#).

Related topics

[Programming Guide for HLSL](#)

Dynamic Linking

Article • 08/23/2019 • 2 minutes to read

Graphics developers sometimes create large, general-purpose shaders that can be used by a wide variety of scene items. At runtime, the shader conditionally runs code appropriate for the given situation. Unfortunately, these large, general-purpose shaders use general-purpose registers (GPRs) inefficiently, and can be much slower than smaller, more targeted shaders.

Shader model 5 addresses this performance problem by introducing dynamic shader linking. Dynamic linking separates shader code fragments by using interfaces and virtual functions and allows the application to select the fragment to use at draw time. This improves performance by binding only the shader code needed and not the entire large, general-purpose shader.

In This Section

Item	Description
Storing Variables and Types for Shaders to Share	Describes the class linkage object for storing variables and types that multiple shaders can share.
Interfaces and Classes	Describes using HLSL interfaces and classes to implement dynamic linking.
Interface Usage Restrictions	Describes restrictions on the use of interfaces in shader code.

Related topics

[HLSL](#)

Storing Variables and Types for Shaders to Share

Article • 08/19/2020 • 2 minutes to read

The class linkage object is a namespace for variables and types that multiple shaders can share. When you pass a class linkage object in a call to create a shader, the runtime gathers a list of variables and types that can implement each interface in the shader and stores the names of those variables and types in the class linkage object.

Therefore, when you call the [ID3D11ClassLinkage::GetClassInstance](#) method to generate class instances from the class linkage object, the runtime can retrieve the variable or type that corresponds to the name that is provided in each shader (if that name is valid for a given shader) and that is created with the given class linkage object.

For example, suppose you have a **Light** class that implements a **Color** interface, and you use this class in your vertex shader and pixel shader. When you create a shader (for example, by calling [ID3D11Device::CreatePixelShader](#)), the runtime determines that the **Light** class type is available in both vertex and pixel shaders and adds the **Light** class type to the class linkage object. You can then create a **Light** instance at a location that you want, bind the resources for both shaders, and pass this instance in the class instances array when you set the shader to the device (for example, by calling [ID3D11DeviceContext::PSSetShader](#)). The runtime then performs the following sequence:

1. Verifies that the instance was created with the same class linkage object.
2. Verifies that the **Light** class type is available in both vertex and pixel shaders.
3. Selects the correct function tables, which can be different for the vertex and pixel shaders.
4. Sends down the offsets that the instance provides.

The class linkage object is ultimately a repository of type and variable names. The maximum number of names available for each item (type and variable) is 64K. The longer the type and variable names are, the higher the storage requirement is for the interface metadata that is stored per shader. This is because the runtime must store a mapping for these names for each shader.

Related Topics

[Dynamic Linking](#)

Related topics

[Dynamic Linking](#)

Interfaces and classes

Article • 08/19/2020 • 4 minutes to read

Dynamic shader linkage makes use of high-level shader language (HLSL) interfaces and classes that are syntactically similar to their C++ counterparts. This allows shaders to reference abstract interface instances at compile time and leave resolution of those instances to concrete classes for the application at runtime.

The following sections detail how to setup a shader to use interfaces and classes and how to initialize interface instances in application code.

- [Declaring Interfaces](#)
- [Declaring Classes](#)
- [Interface Instance Declarations in a Shader](#)
- [Class Instance Declarations in a Shader](#)
- [Initializing Interface Instances in an Application](#)
- [Related topics](#)

Declaring interfaces

An interface functions in a similar manner to an abstract base class in C++. An interface is declared in a shader using the `interface` keyword and only contains method declarations. The methods declared in an interface will all be virtual methods in any classes derived from the interface. Derived classes must implement all methods declared in an interface. Note that interfaces are the only way to declare virtual methods, there is no `virtual` keyword as in C++, and classes cannot declare virtual methods.

The following example shader code declares two interfaces.

```
interface iBaseLight
{
    float3 IlluminateAmbient(float3 vNormal);
    float3 IlluminateDiffuse(float3 vNormal);
    float3 IlluminateSpecular(float3 vNormal, int specularPower );
};

interface iBaseMaterial
{
    float3 GetAmbientColor(float2 vTexcoord);

    float3 GetDiffuseColor(float2 vTexcoord);
```

```
    int GetSpecularPower();  
};
```

Declaring Classes

A class behaves in a similar manner to classes in C++. A class is declared with the class keyword and can contain member variables and methods. A class can inherit from zero or one class and zero or more interfaces. Classes must implement or inherit implementations for all interfaces in its inheritance chain or the class cannot be instantiated.

The following example shader code illustrates deriving a class from an interface and from another class.

```
class cAmbientLight : iBaseLight  
{  
    float3          m_vLightColor;  
    bool      m_bEnable;  
    float3 IlluminateAmbient(float3 vNormal);  
    float3 IlluminateDiffuse(float3 vNormal);  
    float3 IlluminateSpecular(float3 vNormal, int specularPower );  
};  
  
class cHemiAmbientLight : cAmbientLight  
{  
    float4    m_vgroundColor;  
    float4    m_vDirUp;  
    float3 IlluminateAmbient(float3 vNormal);  
};
```

Interface Instance Declarations in a Shader

An interface instance acts as a place holder for class instances that provide an implementation of the interface's methods. Using an instance of an interface allows shader code to call a method without knowing which implementation of that method will be invoked. Shader code declares one or more instances for each interface it defines. These instances are used in shader code in a similar manner to C++ base class pointers.

The following example shader code illustrates declaring several interface instances and using them in shader code.

```
// Declare interface instances
iBaseLight    g_abstractAmbientLighting;
iBaseLight    g_abstractDirectLighting;
iBaseMaterial g_abstractMaterial;

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

float4 PSMain( PS_INPUT Input ) : SV_TARGET
{
    float3 Ambient = (float3)0.0f;
    Ambient = g_abstractMaterial.GetAmbientColor( Input.vTexcoord ) *
        g_abstractAmbientLighting.IlluminateAmbient( Input.vNormal );

    float3 Diffuse = (float3)0.0f;
    Diffuse += g_abstractMaterial.GetDiffuseColor( Input.vTexcoord ) *
        g_abstractDirectLighting.IlluminateDiffuse( Input.vNormal );

    float3 Specular = (float3)0.0f;
    Specular += g_abstractDirectLighting.IlluminateSpecular( Input.vNormal,
        g_abstractMaterial.GetSpecularPower() );

    float3 Lighting = saturate( Ambient + Diffuse + Specular );

    return float4(Lighting,1.0f);
}
```

Class Instance Declarations in a Shader

Each class that will be used in place of an interface instance must either be declared as a variable in a constant buffer or created by the application at runtime using the [ID3D11ClassLinkage::CreateClassInstance](#) method. Interface instances will be pointed at class instances in the application code. Class instances can be referenced in shader code like any other variable, but a class that is derived from an interface will typically only be used with an interface instance and will not be referenced by shader code directly.

The following example shader code illustrates declaring several class instances.

```
cbuffer cbPerFrame : register( b0 )
{
    cAmbientLight      g_ambientLight;
    cHemiAmbientLight  g_hemiAmbientLight;
    cDirectionalLight  g_directionalLight;
    cEnvironmentLight  g_environmentLight;
    float4              g_vEyeDir;
};
```

Initializing Interface Instances in an Application

Interface instances are initialized in application code by passing a dynamic linkage array containing interface assignments to one of the [ID3D11DeviceContext](#) SetShader methods.

To create a dynamic linkage array use the following steps

1. Create a class linkage object using [CreateClassLinkage](#).

```
ID3D11ClassLinkage* g_pPSClassLinkage = NULL;
pd3dDevice->CreateClassLinkage( &g_pPSClassLinkage );
```

2. Create the shader that will be using dynamic class linking, passing the class linkage object as a parameter to the shader's create function.

```
pd3dDevice->CreatePixelShader( pPixelShaderBuffer->GetBufferPointer(),
                                pPixelShaderBuffer->GetBufferSize(), g_pPSClassLinkage,
                                &g_pPixelShader ) ;
```

3. Create a [ID3D11ShaderReflection](#) object using the [D3DReflect](#) function.

```
ID3D11ShaderReflection* pReflector = NULL;
D3DReflect( pPixelShaderBuffer->GetBufferPointer(),
            pPixelShaderBuffer->GetBufferSize(),
            IID_ID3D11ShaderReflection, (void**) &pReflector) ;
```

4. Use the shader reflection object to get the number of interface instances in the shader using the [ID3D11ShaderReflection::GetNumInterfaceSlots](#) method.

```
g_iNumPSInterfaces = pReflector->GetNumInterfaceSlots();
```

5. Create an array large enough to hold the number of interface instances in the shader.

```
ID3D11ClassInstance** g_dynamicLinkageArray = NULL;  
g_dynamicLinkageArray =  
    (ID3D11ClassInstance**) malloc( sizeof(ID3D11ClassInstance*) *  
        g_iNumPSInterfaces );
```

6. Determine the index in the array that corresponds to each interface instance using [ID3D11ShaderReflection::GetVariableByName](#) and [ID3D11ShaderReflectionVariable::GetInterfaceSlot](#).

```
ID3D11ShaderReflectionVariable* pAmbientLightingVar =  
    pReflector->GetVariableByName("g_abstractAmbientLighting");  
    g_iAmbientLightingOffset = pAmbientLightingVar->GetInterfaceSlot(0);
```

7. Get a class instance for each class object derived from an interface in the shader using [ID3D11ClassLinkage::GetInstance](#).

```
g_pPSClassLinkage->GetInstance( "g_hemiAmbientLight", 0,  
    &g_pHemiAmbientLightClass );
```

8. Set interface instances to class instances by setting the corresponding entry in the dynamic linkage array.

```
g_dynamicLinkageArray[g_iAmbientLightingOffset] =  
g_pHemiAmbientLightClass;
```

9. Pass the dynamic linkage array as a parameter to a SetShader call.

```
pd3dImmediateContext->PSSetShader( g_pPixelShader,  
g_dynamicLinkageArray, g_iNumPSIInterfaces );
```

Related topics

[Dynamic Linking](#)

Interface Usage Restrictions

Article • 08/23/2019 • 2 minutes to read

Current GPU hardware does not support varying slot information at shader runtime. As a consequence interface references cannot be modified within a conditional expression such as an if or switch statement.

The following shader code illustrates when this restriction will occur and a possible alternate approach.

Given the following interface declarations:

```
interface A
{
    float GetRatio();
    bool IsGood();
};

interface B
{
    float GetValue();
};

A arrayA[6];
B arrayB[6];
```

Given the following class declarations:

```
class C1 : A
{
    float var;
    float GetRatio() { return 1.0f; }
    bool IsGood() { return true; }
};

class C2 : C1, B
{
    float GetRatio() { return C1::GetRatio() * 0.33f; }
    float GetValue() { return 5.0f; }
    bool IsGood() { return false; }
};

class C3 : B
{
    float var;
```

```

    float GetValue() { return -1.0f; }

};

class C4 : A, B
{
    float var;
    float GetRatio() { return var; }
    float GetValue() { return var * 2.0f; }
    bool IsGood() { return var > 0.0f; }
};

```

An interface reference cannot be modified within the conditional expression (an if statement):

```

float main() : wicked
{
    float rev;
    {
        A a = arrayA[0];
        for( uint i = 0; i < 6; ++i )
        {
            if( arrayA[i].IsGood() )
            {
                // This forces the loop to be unrolled,
                // since the slot information is changing.
                a = arrayA[i];
                rev -= arrayA[i-2].GetRatio();
            }
            else
            {
                // This causes an error since the compiler is
                // unable to determine the interface slot
                rev += arrayB[i].GetValue() + a.GetRatio();
            }
        }
    }
    return rev;
}

```

Given the same interface and class declarations, you could use an index to provide the same functionality and avoid the forced loop unroll.

```

float main() : wicked
{
    float rev;
    {
        uint index = 0;

```

```
for( uint i = 0; i < 6; ++i )
{
    if( arrayA[i].IsGood() )
    {
        index = i;
        rev -= arrayA[i-2].GetRatio();
    }
    else
    {
        rev += arrayB[i].GetValue() + arrayA[index].GetRatio();
    }
}
return rev;
}
```

Related topics

[Dynamic Linking](#)

Geometry Shader Features

Article • 08/23/2019 • 2 minutes to read

This section contains overview material for new geometry shader features introduced in shader model 5.

In This Section

Item	Description
How To: Index Multiple Output Streams	Use indexing to declare up to four output streams in a single geometry shader.
How To: Instance a Geometry Shader	Execute one or more instances of a geometry shader per primitive.

Related topics

[HLSL Shader Model 5](#)

How To: Index Multiple Output Streams

Article • 04/26/2022 • 2 minutes to read

In shader model 5, a geometry shader can support up to 4 separate streams. This means a single shader can output between one and four output streams, depending on the number of streams declared.

To index multiple output streams

1. Define a data stream using a stream template type.

```
inout PointStream<OutVertex1> myStream1,
```

2. Define a second data stream using a stream template type.

```
inout PointStream<OutVertex2> myStream2 )
```

3. Output data to either (or both) streams using the stream output object intrinsic functions (such as Append or RestartStrip).

```
void MyGS(
    InVertex verts[2],
    inout PointStream<OutVertex1> myStream1,
    inout PointStream<OutVertex2> myStream2 )
{
    OutVertex1 myVert1 = TransformVertex1( verts[0] );
    OutVertex2 myVert2 = TransformVertex2( verts[1] );
    myStream1.Append( myVert1 );
    myStream2.Append( myVert2 );
}
```

When using a single output stream, you can emit triangle strips, line strips, or point lists. When you store triangle and line strips in the stream out buffer, they are expanded to triangle and line lists respectively. You can also rasterize one stream and not send it to a memory buffer.

When using multiple output streams, all streams must contain points, and up to one output stream can be sent to the rasterizer. More commonly, an application will not

rasterize any stream.

After you stream data to a buffer, you can use that data to render any primitive type, not just the primitive type that you used to fill the buffer.

The total output of the geometry shader is limited to 1024 scalars. When multiple streams exist, the number of scalars is computed from the largest stream type multiplied by the maximum vertex count.

Differences between shader model 4 and shader model 5:

Shader model 4:

- Maximum number of scalars for stream output is 64.
- The per-component register mask must match across the index range.

Shader model 5:

- Maximum number of scalars for stream output is 128.
- The per-component register mask does not need to match across the index range.
- Dynamic indexing of outputs must be legal across all streams.
- Interpolation modes do not need to match for the streams.

Related topics

[Geometry Shader Features](#)

How To: Instance a Geometry Shader

Article • 08/23/2019 • 2 minutes to read

Geometry shader instancing allows multiple executions of the same geometry shader to be executed per primitive. To instance a geometry shader, add an instance attribute to the main shader function and identify an instance index parameter in the shader function body.

To Instance a Geometry Shader:

1. Add the [instance attribute](#) to the main function.

```
[instance(24)]
```

This defines the number of instances (a maximum of 32) to be run for each primitive.

2. Attach the [SV_GSInstanceID](#) system value to a variable in the function parameter list that can be used to track the ID of the instance being executed.

```
uint InstanceID : SV_GSInstanceID
```

3. Compile and create the shader just as you would any other geometry shader.

Other details include:

- The maximum instance count is 32.
- The maximum vertex count is a per-instance maximum vertex count.
- Each instance invocation (like any geometry shader invocation) increases the invocation count and generates an implicit `RestartStrip()`.

Related topics

[Geometry Shader Features](#)

HLSL Shader Model 5.1

Article • 08/19/2020 • 2 minutes to read

This section describes the features of Shader Model 5.1 as they apply in practice to D3D12. All DirectX 12 hardware supports Shader Model 5.1.

In this section

Topic	Description
Bytecode changes in SM5.1	SM5.1 changes how resource registers are declared and referenced in instructions.
HLSL Specified Root Signature	A Root Signature (a key table of resources and other elements for D3D12) can be specified in HLSL as a string.

For details of the syntax changes to the shader language, refer to [Shader Model 5.1](#).

Related topics

[Programming Guide for HLSL](#)

Bytecode changes in SM5.1

Article • 08/23/2019 • 3 minutes to read

SM5.1 changes how resource registers are declared and referenced in instructions.

SM5.1 moves towards declaring a register “variable”, similar to how it is done for group shared memory registers, illustrated by the following example:

```
syntax

Texture2D<float4> tex0      : register(t5,  space0);
Texture2D<float4> tex1[][][3] : register(t10, space0);
Texture2D<float4> tex2[8]     : register(t0,   space1);
SamplerState samp0           : register(s5,  space0);

float4 main(float4 coord : COORD) : SV_TARGET
{
    float4 r = coord;
    r += tex0.Sample(samp0, r.xy);
    r += tex2[r.x].Sample(samp0, r.xy);
    r += tex1[r.x][r.y][r.z].Sample(samp0, r.xy);
    return r;
}
```

The disassembly of this example follows:

```
syntax

// Resource Bindings:
//
// Name          Type  Format      Dim Space Slot
Elements
// -----
-----
// samp0         sampler    NA        NA      0      5
1
// tex0          texture   float4     2d      0      5
1
// tex1[0][5][3] unbounded
// tex2[8]        texture   float4     2d      1      0
8
//
//
//
// Input signature:
//
// Name          Index  Mask Register SysValue Format Used
// -----
// COORD         0      xyzw       0      NONE   float   xyzw
```

```

//  

//  

// Output signature:  

//  

// Name           Index  Mask Register SysValue Format  Used  

// -----  

// SV_TARGET      0      xyzw       0      TARGET   float   xyzw  

//  

ps_5_1  

dcl_globalFlags refactoringAllowed  

dcl_sampler s0[5:5], mode_default, space=0  

dcl_resource_texture2d (float,float,float,float) t0[5:5], space=0  

dcl_resource_texture2d (float,float,float,float) t1[10:*], space=0  

dcl_resource_texture2d (float,float,float,float) t2[0:7], space=1  

dcl_input_ps linear v0.xyzw  

dcl_output o0.xyzw  

dcl_temps 2  

sample r0.xyzw, v0.xyxx, t0[0].xyzw, s0[5]  

add r0.xyzw, r0.xyzw, v0.xyzw  

ftou r1.x, r0.x  

sample r1.xyzw, r0.xyxx, t2[r1.x + 0].xyzw, s0[5]  

add r0.xyzw, r0.xyzw, r1.xyzw  

ftou r1.xyz, r0.zyxz  

imul null, r1.yz, r1.zzyz, l(0, 15, 3, 0)  

iadd r1.y, r1.z, r1.y  

iadd r1.x, r1.x, r1.y  

sample r1.xyzw, r0.xyxx, t1[r1.x + 10].xyzw, s0[5]  

add o0.xyzw, r0.xyzw, r1.xyzw  

ret  

// Approximately 12 instruction slots used

```

Each shader resource range now has an ID (a name) in the shader bytecode. For example, tex1 texture array becomes 't1' in the shader byte code. Giving unique IDs to each resource range allows two things:

- Unambiguously identify which resource range (see `dcl_resource_texture2d`) is being indexed in an instruction (see sample instruction).
- Attach set of attributes to the declaration, e.g., element type, stride size, raster operation mode, etc..

Note that the ID of the range is not related to the HLSL lower bound declaration.

The order of reflection resource bindings and shader declaration instructions is the same to aid in identifying the correspondence between HLSL variables and bytecode IDs.

Each declaration instruction in SM5.1 uses a 3D operand to define: range ID, lower and upper bounds. An additional token is emitted to specify the register space. Other tokens may be emitted as well to convey additional properties of the range, e.g., cbuffer or structured buffer declaration instruction emits the size of the cbuffer or structure. The

exact details of encoding can be found in `d3d12TokenizedProgramFormat.h` and `D3D10ShaderBinary::CShaderCodeParser`.

SM5.1 instructions will not emit additional resource operand information as part of the instruction (as in SM5.0). This information is now moved to the declaration instructions. In SM5.0, instructions indexing resources required resource attributes to be described in extended opcode tokens, since indexing obfuscated the association to the declaration. In SM5.1 each ID (such as 't1') is unambiguously associated with a single declaration that describes the required resource information. Therefore, the extended opcode tokens used on instructions to describe resource information are no longer emitted.

In non-declaration instructions, a resource operand for samplers, SRVs, and UAVs is a 2D operand. The first index is a literal constant that specifies the range ID. The second index represents the linearized value of the index. The value is computed relative to the beginning of the corresponding register space (not relative to the beginning of the logical range) to better correlate with the root signature and to reduce the driver compiler burden of adjusting the index.

A resource operand for CBVs is a 3D operand: literal ID of the range, index of the cbuffer, offset into the particular instance of cbuffer.

Related topics

[HLSL Shader Model 5.1 Features for Direct3D 12](#)

[Shader Model 5.1](#)

HLSL Specified Root Signature

Article • 08/19/2020 • 2 minutes to read

A [Root Signature](#) (a key table of resources and other elements for D3D12) can be specified in HLSL as a string.

Refer to the D3D12 docs on [Specifying Root Signatures in HLSL](#) for full details and example code.

Related topics

[HLSL Shader Model 5.1 Features for Direct3D 12](#)

[Shader Model 5.1](#)

HLSL Shader Model 6.0

Article • 08/25/2021 • 6 minutes to read

Describes the wave operation intrinsics added to HLSL Shader Model 6.0.

- [Shader model 6.0](#)
- [Terminology](#)
- [Shading language intrinsics](#)
 - [Wave Query](#)
 - [Wave Vote](#)
 - [Wave Broadcast](#)
 - [Wave Reduction](#)
 - [Wave Scan and Prefix](#)
 - [Quad-wide Shuffle operations](#)
- [Hardware capability](#)
- [Related topics](#)

Shader Model 6.0

For earlier shader models, HLSL programming exposes only a single thread of execution. New wave-level operations are provided, starting with model 6.0, to explicitly take advantage of the parallelism of current GPUs - many threads can be executing in lockstep on the same core simultaneously. For example, the model 6.0 intrinsics enable the elimination of barrier constructs when the scope of synchronization is within the width of the SIMD processor, or some other set of threads that are known to be atomic relative to each other.

Potential use cases include: stream compaction, reductions, block transpose, bitonic sort or Fast Fourier Transforms (FFT), binning, stream de-duplication, and similar scenarios.

Most of the intrinsics appear in pixel shaders and compute shaders, though there are some exceptions (noted for each function). The functions have been added to the requirements for DirectX Feature Level 12.0, under API level 12.

The `<type>` parameter and return value for these functions implies the type of the expression, the supported types are those from the following list that are *also* present in the target shader model for your app:

- `half, half2, half3, half4`
- `float, float2, float3, float4`
- `double, double2, double3, double4`

- int, int2, int3, int4
- uint, uint2, uint3, uint4
- short, short2, short3, short4
- ushort, ushort2, ushort3, ushort4
- uint64_t, uint64_t2, uint64_t3, uint64_t4

Some operations (such as the bitwise operators) only support the integer types.

Terminology

Term	Definition
Lane	A single thread of execution. The shader models before version 6.0 expose only one of these at the language level, leaving expansion to parallel SIMD processing entirely up to the implementation.
Wave	A set of lanes (threads) executed simultaneously in the processor. No explicit barriers are required to guarantee that they execute in parallel. Similar concepts include "warp" and "wavefront."
Inactive Lane	A lane which is not being executed, for example due to the flow of control, or insufficient work to fill the minimum size of the wave.
Active Lane	A lane for which execution is being performed. In pixel shaders, it may include any helper pixel lanes.
Quad	A set of 4 adjacent lanes corresponding to pixels arranged in a 2x2 square. They are used to estimate gradients by differencing in either x or y. A wave may be comprised of multiple quads. All pixels in an active quad are executed (and may be "Active Lanes"), but those that do not produce visible results are termed "Helper Lanes".
Helper Lane	A lane which is executed solely for the purpose of gradients in pixel shader quads. The output of such a lane will be discarded, and so not render to the destination surface.

Shading language intrinsics

All the operations of this shader model have been added in a range of intrinsic functions.

Wave Query

The intrinsics for querying a single wave.

Intrinsic	Description	Pixel shader	Compute shader

Intrinsic	Description	Pixel shader	Compute shader
WaveGetLaneCount	Returns the number of lanes in the current wave.	*	*
WaveGetLaneIndex	Returns the index of the current lane within the current wave.	*	*
WavesFirstLane	Returns true only for the active lane in the current wave with the smallest index	*	*

Wave Vote

This set of intrinsics compare values across threads currently active from the current wave.

Intrinsic	Description	Pixel shader	Compute shader
WaveActiveAnyTrue	Returns true if the expression is true in any active lane in the current wave.	*	*
WaveActiveAllTrue	Returns true if the expression is true in all active lanes in the current wave.	*	*
WaveActiveBallot	Returns a 64-bit unsigned integer bitmask of the evaluation of the Boolean expression for all active lanes in the specified wave.	*	*

Wave Broadcast

These intrinsics enable all active lanes in the current wave to receive the value from the specified lane, effectively broadcasting it. The return value from an invalid lane is undefined.

Intrinsic	Description	Pixel shader	Compute shader
WaveReadLaneAt	Returns the value of the expression for the given lane index within the specified wave.	*	*
WaveReadLaneFirst	Returns the value of the expression for the active lane of the current wave with the smallest index.	*	*

Wave Reduction

These intrinsics compute the specified operation across all active lanes in the wave and broadcast the final result to all active lanes. Therefore, the final output is guaranteed uniform across the wave.

Intrinsic	Description	Pixel shader	Compute shader
WaveActiveAllEqual	Returns true if the expression is the same for every active lane in the current wave (and thus uniform across it).	*	*
WaveActiveBitAnd	Returns the bitwise AND of all the values of the expression across all active lanes in the current wave, and replicates the result to all lanes in the wave.	*	*
WaveActiveBitOr	Returns the bitwise OR of all the values of the expression across all active lanes in the current wave, and replicates the result to all lanes in the wave.	*	*
WaveActiveBitXor	Returns the bitwise Exclusive OR of all the values of the expression across all active lanes in the current wave, and replicates the result to all lanes in the wave.	*	*
WaveActiveCountBits	Counts the number of boolean variables which evaluate to true across all active lanes in the current wave, and replicates the result to all lanes in the wave.	*	*
WaveActiveMax	Computes the maximum value of the expression across all active lanes in the current wave, and replicates the result to all lanes in the wave.	*	*
WaveActiveMin	Computes the minimum value of the expression across all active lanes in the current wave, and replicates the result to all lanes in the wave.	*	*
WaveActiveProduct	Multiplies the values of the expression together across all active lanes in the current wave, and replicates the result to all lanes in the wave.	*	*
WaveActiveSum	Sums up the value of the expression across all active lanes in the current wave and replicates it to all lanes in the current wave, and replicates the result to all lanes in the wave.	*	*

Wave Scan and Prefix

These intrinsics apply the operation to each lane and leave each partial result of the computation in the corresponding lane.

Intrinsic	Description	Pixel shader	Compute shader
WavePrefixCountBits	Returns the sum of all the specified boolean variables set to true across all active lanes with indices smaller than the current lane.	*	*
WavePrefixSum	Returns the sum of all of the values in the active lanes with smaller indices than this one.	*	*
WavePrefixProduct	Returns the product of all of the values in the lanes before this one of the specified wave.	*	*

Quad-wide Shuffle operations

These intrinsics perform swap operations on the values across a wave known to contain pixel shader quads as defined here. The indices of the pixels in the quad are defined in scan-line or reading order - where the coordinates within a quad are:

+-----> X

| [0] [1]

| [2] [3]

v

Y

These routines work in either compute shaders or pixel shaders. In compute shaders they operate in quads defined as evenly divided groups of 4 within an SIMD wave. In pixel shaders they should be used on waves captured by WaveQuadLanes, otherwise results are undefined.

Intrinsic	Description	Pixel shader	Compute shader
QuadReadLaneAt	Returns the specified source value read from the lane of the current quad identified by quadLaneID [0..3] which must be uniform across the quad.	*	
QuadReadAcrossDiagonal	Returns the specified local value which is read from the diagonally opposite lane in this quad.		*

Intrinsic	Description	Pixel shader	Compute shader
QuadReadAcrossX	Returns the specified source value read from the other lane in this quad in the X direction.	*	
QuadReadAcrossY	Returns the specified source value read from the other lane in this quad in the Y direction.	*	

Hardware capability

In order to check that the wave operation features are available on any specific hardware, call [ID3D12Device::CheckFeatureSupport](#), noting the description and use of the [D3D12_FEATURE_DATA_D3D12_OPTIONS1](#) structure.

Related topics

- [Programming Guide for HLSL](#)
- [Shader Model 6 intrinsics](#)

HLSL Shader Model 6.4

Article • 03/05/2021 • 2 minutes to read

Describes the machine learning intrinsics added to HLSL Shader Model 6.4.

Shader Model 6.4

These intrinsics are a required/supported feature of Shader model 6.4. Consequently, no separate capability bit check is required, beyond assuring the use of Shader Model 6.4. The minimum supported client for these routines is Windows 10, version 1903.

Shading language intrinsics

Unsigned Integer Dot-Product of 4 Elements and Accumulate

syntax

```
uint32 dot4add_u8packed(uint32 a, uint32 b, uint32 acc); // ubyte4 a, b;
```

A 4-dimensional unsigned integer dot-product with add. Multiplies together each corresponding pair of unsigned 8-bit int bytes in the two input DWORDs, and sums the results into the 32-bit unsigned integer accumulator. This instruction operates within a single 32-bit wide SIMD lane. The inputs are also assumed to be 32-bit quantities.

Signed Integer Dot-Product of 4 Elements and Accumulate

syntax

```
int32 dot4add_i8packed(uint32 a, uint32 b, int32 acc); // signed byte4 a, b;
```

A 4-dimensional signed integer dot-product with add. Multiplies together each corresponding pair of signed 8-bit int bytes in the two input DWORDs, and sums the results into the 32-bit signed integer accumulator. This instruction operates within a single 32-bit wide SIMD lane. The inputs are also assumed to be 32-bit quantities.

Single Precision Floating Point 2-Element Dot-Product and Accumulate

syntax

```
float dot2add( half2 a, half2 b, float acc );
```

A 2-dimensional floating point dot-product of half2 vectors with add. Multiplies the elements of the two half-precision float input vectors together and sums the results into the 32-bit float accumulator. This instruction operates within a single 32-bit wide SIMD lane. The inputs are 16-bit quantities packed into the same lane.

This is covered under the low-precision feature bit (indicating that native half and short support are present).

SV_ShadingRate

syntax

```
uint shadingRate : SV_ShadingRate
```

An unsigned integer representing how many target pixels are written by each invocation of the pixel shader. Valid values belong to set of enumeration values [D3D12_SHADING_RATE](#).

This system value is available on platforms that are [D3D12_VARIABLE_SHADING_RATE_TIER_2](#) or higher. It can be written from at most one of vertex or geometry shader stages. It can be read from the pixel shader stage. For more information, see the [Variable-rate Shading](#).

Reference for HLSL

Article • 08/23/2019 • 2 minutes to read

The HLSL reference documentation specifies the language characteristics. It is broken into several sections.

- [Language Syntax \(DirectX HLSL\)](#) - Programming shaders in HLSL requires that you understand the language syntax, that is, how you write HLSL code. This includes code to declare and initialize variables, write user-defined shader functions, and add flow control statements to make your functions more powerful.
- [Shader Models vs Shader Profiles](#) - The HLSL compiler implements rules and restrictions based on shader models. The code in each vertex shader, geometry shader (if you are using Direct3D 10) and pixel shader are validated against a shader model, which you supply at compile time.
- [Intrinsic Functions \(DirectX HLSL\)](#) - HLSL has many intrinsic functions. These are implemented and tested so that you can use them knowing that they are already debugged and they perform well. If you choose to write your own functions, see the language syntax section for writing user-defined functions.
- [Asm Shader Reference](#) - Assembly instructions that you can use to program and debug shaders.
- [D3DCompiler Reference](#) - Compiles raw HLSL source.
- [Inline Format Conversion Reference](#) - The D3DX_DXGIFormatConvert.inl file contains inline format conversion functions that you can use in the compute shader or pixel shader on Direct3D 11 hardware. You can use these functions in your application to simultaneously both read from and write to a texture. That is, you can perform in-place image editing. To use these inline format conversion functions, include the D3DX_DXGIFormatConvert.inl file in your application.
- [Appendix \(DirectX HLSL\)](#) - The appendix is included for completeness. It includes a listing of the keywords and reserved words; these words cannot be used as identifiers in your programs. It also includes a listing of the language grammar for reference.
- [HLSL errors and warnings](#) - Provides error and warning codes that a shader can return.

Related topics

[HLSL](#)

[Programming Guide for HLSL](#)

Language Syntax

Article • 08/23/2019 • 2 minutes to read

HLSL shaders are made up of variables, and functions, which in turn are made up of statements. The language syntax documents how to define and declare variables, add flow control so that shaders can make runtime decisions based on variables, and write custom functions.

- [Variables](#)
- [Flow Control](#)
- [Functions](#)
- [Statements](#)
- [Semantics](#)

Related topics

[Reference for HLSL](#)

Variables

Article • 08/23/2019 • 2 minutes to read

HLSL variables are similar to variables defined in the C programming language. Similar to C, variables have some naming restrictions, have scoping properties that depend on where they are declared, and can have user metadata attached to them. Like C, there are several standard data types. Unlike C, there are also additional data types defined by HLSL to help maximize the performance of 4-component vectors that use matrix math to operate on 3D graphics data.

- [Variable Syntax](#)
- [Data Types](#)
- [Semantics](#)

Related topics

[Language Syntax \(DirectX HLSL\)](#)

Variable Syntax

Article • 08/19/2021 • 6 minutes to read

Use the following syntax rules to declare HLSL variables.

[*Storage_Class*] [*Type_Modifier*] *Type Name[Index]* [: *Semantic*] [: *Packoffset*] [: *Register*];
[*Annotations*] [= *Initial_Value*]

Parameters

Storage_Class

Optional storage-class modifiers that give the compiler hints about variable scope and lifetime; the modifiers can be specified in any order.

Value	Description
extern	Mark a global variable as an external input to the shader; this is the default marking for all global variables. Cannot be combined with static .
nointerpolation	Do not interpolate the outputs of a vertex shader before passing them to a pixel shader.

Value	Description
precise	<p>The precise keyword when applied to a variable will restrict any calculations used to produce the value assigned to that variable in the following ways:[*] Separate operations are kept separate. For example, where a mul and add operation might have been fused into a mad operation, precise forces the operations to remain separate. Instead, you must explicitly use the mad intrinsic function.* Order of operations are maintained. Where the order of instructions might have been shuffled to improve performance, precise ensures that the compiler preserves the order as written.* IEEE unsafe operations are restricted. Where the compiler might have used fast math operations that don't account for NaN (not a number) and INF (infinite) values, precise forces IEEE requirements concerning NaN and INF values to be respected. Without precise, these optimizations and mathematical operations are not IEEE safe.* Qualifying a variable precise doesn't make operations that use the variable precise. Since precise propagates only to operations that contribute to the values that are assigned to the precise-qualified variable, correctly making desired calculations precise can be tricky, so we recommended that you mark the shader outputs precise directly where you declare them, whether that's on a structure field, or on an output parameter, or the return type of the entry function. The ability to control optimizations in this way maintains result invariance for the modified output variable by disabling optimizations that might affect final results due to differences in accumulated precision differences. It is useful when you want shaders for tessellation to maintain water-tight patch seams or match depth values over multiple passes.</p> <p>Sample code ↗: HLSLmatrix</p> <pre data-bbox="414 1147 1361 1304">g_mWorldViewProjection;void main(in float3 InPos : Position, out precise float4 OutPos : SV_Position){ // operation is precise because it contributes to the precise parameter OutPos OutPos = mul(float4(InPos, 1.0), g_mWorldViewProjection);}</pre>
shared	Mark a variable for sharing between effects; this is a hint to the compiler.
groupshared	Mark a variable for thread-group-shared memory for compute shaders. In D3D10 the maximum total size of all variables with the groupshared storage class is 16kb, in D3D11 the maximum size is 32kb. See examples.
static	Mark a local variable so that it is initialized one time and persists between function calls. If the declaration does not include an initializer, the value is set to zero. A global variable marked static is not visible to an application.
uniform	Mark a variable whose data is constant throughout the execution of a shader (such as a material color in a vertex shader); global variables are considered uniform by default.

Value	Description
volatile	Mark a variable that changes frequently; this is a hint to the compiler. This storage class modifier only applies to a local variable. [!Note] The HLSL compiler currently ignores this storage class modifier.

Type_Modifier

Optional variable-type modifier.

Value	Description
const	Mark a variable that cannot be changed by a shader, therefore, it must be initialized in the variable declaration. Global variables are considered const by default (suppress this behavior by supplying the /Gec flag to the compiler).
row_major	Mark a variable that stores four components in a single row so they can be stored in a single constant register.
column_major	Mark a variable that stores 4 components in a single column to optimize matrix math.

① Note

If you do not specify a type-modifier value, the compiler uses **column_major** as the default value.

Type

Any HLSL type listed in [Data Types \(DirectX HLSL\)](#).

Name[Index]

ASCII string that uniquely identifies a shader variable. To define an optional array, use **index** for the array size, which is a positive integer = 1.

Semantic

Optional parameter-usage information, used by the compiler to link shader inputs and outputs. There are several predefined **semantics** for vertex and pixel shaders. The compiler ignores semantics unless they are declared on a global variable, or a parameter passed into a shader.

Packoffset

Optional keyword for manually packing shader constants. See [packoffset \(DirectX HLSL\)](#).

Register

Optional keyword for manually assigning a shader variable to a particular register. See [register \(DirectX HLSL\)](#).

Annotation(s)

Optional metadata, in the form of a string, attached to a global variable. An annotation is used by the effect framework and ignored by HLSL; to see more detailed syntax, see [annotation syntax](#).

Initial_Value

Optional initial value(s); the number of values should match the number of components in *Type*. Each global variable marked **extern** must be initialized with a literal value; each variable marked **static** must be initialized with a constant.

Global variables that are not marked **static** or **extern** are not compiled into the shader. The compiler does not automatically set default values for global variables and cannot use them in optimizations. To initialize this type of global variable, use reflection to get its value and then copy the value to a constant buffer. For example, you can use the [ID3D11ShaderReflection::GetVariableByName](#) method to get the variable, use the [ID3D11ShaderReflectionVariable::GetDesc](#) method to get the shader-variable description, and get the initial value from the **DefaultValue** member of the [D3D11_SHADER_VARIABLE_DESC](#) structure. To copy the value to the constant buffer, you must ensure that the buffer was created with CPU write access ([D3D11_CPU_ACCESS_WRITE](#)). For more information about how to create a constant buffer, see [How to: Create a Constant Buffer](#).

You can also use the [effects framework](#) to automatically process the reflecting and setting the initial value. For example, you can use the [ID3DX11EffectPass::Apply](#) method.

Examples

Here are several examples of shader-variable declarations.

```
float fVar;
```

```
float4 color;
float fVar = 3.1f;

int iVar[3];

int iVar[3] = {1,2,3};

uniform float4 position : SV_POSITION;
const float4 lightDirection = {0,0,1};
```

Group Shared

HLSL enables threads of a compute shader to exchange values via shared memory. HLSL provides barrier primitives such as [GroupMemoryBarrierWithGroupSync](#), and so on to ensure the correct ordering of reads and writes to shared memory in the shader and to avoid data races.

ⓘ Note

Hardware executes threads in groups (warps or wave-fronts), and barrier synchronization can sometimes be omitted to increase performance when only synchronizing threads that belong to the same group is correct. But we highly discourage this omission for these reasons:

- This omission results in non-portable code, which might not work on some hardware and doesn't work on software rasterizers that typically execute threads in smaller groups.
- The performance improvements that you might achieve with this omission will be minor compared to using all-thread barrier.

In Direct3D 10 there is no synchronization of threads when writing to **groupshared**, so this means that each thread is limited to a single location in an array for writing. Use the [SV_GroupIndex](#) system value to index into this array when writing to ensure that no two threads can collide. In terms of reading, all threads have access to the entire array for reading.

```
struct GSData
{
    float4 Color;
```

```
    float Factor;  
}  
  
groupshared GSData data[5*5*1];  
  
[numthreads(5,5,1)]  
void main( uint index : SV_GroupIndex )  
{  
    data[index].Color = (float4)0;  
    data[index].Factor = 2.0f;  
    GroupMemoryBarrierWithGroupSync();  
    ...  
}
```

Packing

Pack subcomponents of vectors and scalars whose size is large enough to prevent crossing register boundaries. For example, these are all valid:

```
cbuffer MyBuffer  
{  
    float4 Element1 : packoffset(c0);  
    float1 Element2 : packoffset(c1);  
    float1 Element3 : packoffset(c1.y);  
}
```

Cannot mix packing types.

Like the register keyword, a packoffset can be target specific. Subcomponent packing is only available with the packoffset keyword, not the register keyword. Inside a cbuffer declaration, the register keyword is ignored for Direct3D 10 targets as it is assumed to be for cross-platform compatibility.

Packed elements may overlap and the compiler will give no error or warning. In this example, Element2 and Element3 will overlap with Element1.x and Element1.y.

```
cbuffer MyBuffer  
{  
    float4 Element1 : packoffset(c0);  
    float1 Element2 : packoffset(c0);  
    float1 Element3 : packoffset(c0.y);  
}
```

A sample that uses packoffset is: [HLSLWithoutFX10 Sample](#).

Related topics

[Variables \(DirectX HLSL\)](#)

packoffset

Article • 06/08/2021 • 2 minutes to read

Optional shader constant packing keyword, which uses the following syntax:

```
: packoffset( c[Subcomponent][.component] )
```

Parameters

Item	Description
packoffset	Required keyword.
c	Packing applies to constant registers (c) only.
[Subcomponent]	Optional subcomponents and components. A subcomponent is a register number, which is an integer. A component is in the form of [.xyzw].
[.component]	

Remarks

Use this keyword to manually pack a shader constant when [declaring a variable type](#).

When packing a constant, you cannot mix constant types.

The compiler behaves slightly differently for global constants and uniform constants:

- A global constant. A global variable is added as a global constant to a *\$Global* cbuffer by the compiler. Automatically packed elements (those declared without packoffset) will appear after the last manually packed variable. You may mix types when packing global constants.
- A uniform constant. A uniform parameter in the parameter list of a function will be added to a *\$Param* constant buffer by the compiler when the shader is compiled outside of the effects framework. When compiled inside the effect framework, a uniform constant must resolve to a uniform variable defined in global scope. A uniform constant cannot be manually offset; their recommend use is only for specialization of shaders where they alias back to globals, not as a means of passing application data into the shader.

Here are some additional examples: [packing constants using shader model 4](#).

Examples

Here are several examples of manually packing shader constants.

Pack subcomponents of vectors and scalars whose size is large enough to prevent crossing register boundaries. For example, these are all valid:

```
cbuffer MyBuffer
{
    float4 Element1 : packoffset(c0);
    float1 Element2 : packoffset(c1);
    float1 Element3 : packoffset(c1.y);
}
```

See also

[Variable Syntax](#)

[Variables \(DirectX HLSL\)](#)

register

Article • 08/19/2020 • 2 minutes to read

Optional keyword for assigning a shader variable to a particular register, which uses the following syntax:

```
:register ([shader_profile], Type#[subcomponent])
```

Parameters

register

Required keyword.

[*shader_profile*]

Optional [shader profile](#), which can be a shader target or simply **ps** or **vs**.

Type#[subcomponent]

Register type, number, and subcomponent declaration.

- Type is one of the following:

Type	Register Description
b	Constant buffer
t	Texture and texture buffer
c	Buffer offset
s	Sampler
u	Unordered Access View

- # is the register number, which is an integer number.
- The *subcomponent* is an optional integer number.

Remarks

You may add one or more register assignments to the same variable declaration, separated by spaces.

For Direct3D 10 variables in global scope, the **register** keyword acts the same as the [packoffset \(DirectX HLSL\)](#) keyword.

Examples

Here are some examples:

```
sampler myVar : register( ps_5_0, s );
```

```
sampler myVar : register( vs, s[8] );
```

```
sampler myVar : register( ps, s[2] )
    : register( ps_5_0, s[0] )
    : register( vs, s[8] );
```

See also

[Variable Syntax](#)

[Variables \(DirectX HLSL\)](#)

Data Types (HLSL)

Article • 06/30/2021 • 2 minutes to read

HLSL supports many different intrinsic data types. This table shows which types to use to define shader variables.

Use this intrinsic type	To define this shader variable
Scalar	One-component scalar
Vector, Matrix	Multiple-component vector or matrix
Sampler, Texture or Buffer	Sampler, texture, or buffer object
Struct, User Defined	Custom structure or typedef
Array	Literal scalar expressions declared containing most other types
State Object	HLSL representations of state objects

To help you better understand how to use vectors and matrices in HLSL, you may want to read this background information on how HLSL uses [per-component](#) math.

Related topics

[Variables \(DirectX HLSL\)](#)

Buffer type

Article • 06/15/2022 • 2 minutes to read

Use the following syntax to declare a buffer variable.

```
Buffer<Type> Name;
```

Parameters

Buffer

Required keyword.

Type

One of the [scalar](#), [vector](#), and some [matrix](#) HLSL types. You can declare a buffer variable with a matrix as long as it fits in 4 32-bit quantities. So, you can write `Buffer<float2x2>`. But `Buffer<float4x4>` is too large, and the compiler will generate an error.

Name

An ASCII string that uniquely identifies the variable name.

Example

Here is an example of a buffer declaration.

```
Buffer<float4> g_Buffer;
```

Data is read from a buffer using an overloaded version of the [Load](#) HLSL intrinsic function that takes one input parameter (an integer index). A buffer is accessed like an array of elements; therefore, this example reads the second element.

```
float4 bufferData = g_Buffer.Load( 1 );
```

Use the [stream-output stage](#) to output data to a buffer.

Remarks

A compatible typed buffer shader resource view (SRV) is required to correctly load from the buffer. The load can optionally perform a type conversion, for example an **RGBA8_UNORM** buffer can be loaded into a `float4` variable. For a buffer containing structs, use a [StructuredBuffer](#) instead.

See also

[Data Types \(DirectX HLSL\)](#)

Scalar Types

Article • 08/20/2021 • 2 minutes to read

HLSL supports several scalar data types:

- **bool** - true or false.
- **int** - 32-bit signed integer.
- **uint** - 32-bit unsigned integer.
- **dword** - 32-bit unsigned integer.
- **half** - 16-bit floating point value. This data type is provided only for language compatibility. Direct3D 10 shader targets map all half data types to float data types. A half data type cannot be used on a uniform global variable (use the /Gec flag if this functionality is desired).
- **float** - 32-bit floating point value.
- **double** - 64-bit floating point value. You cannot use double precision values as inputs and outputs for a stream. To pass double precision values between shaders, declare each **double** as a pair of **uint** data types. Then, use the **asuint** function to pack each **double** into the pair of **uints** and the **asdoule** function to unpack the pair of **uints** back into the **double**.

Starting with Windows 8 HLSL also supports minimum precision scalar data types. Graphics drivers can implement minimum precision scalar data types by using any precision greater than or equal to their specified bit precision. We recommend not to rely on clamping or wrapping behavior that depends on specific underlying precision. For example, the graphics driver might execute arithmetic on a **min16float** value at full 32-bit precision.

- **min16float** - minimum 16-bit floating point value.
- **min10float** - minimum 10-bit floating point value.
- **min16int** - minimum 16-bit signed integer.
- **min12int** - minimum 12-bit signed integer.
- **min16uint** - minimum 16-bit unsigned integer.

For more information about scalar literals, see [Grammar](#).

Differences between Direct3D 9 and Direct3D 10:

In Direct3D 10, the following types are modifiers to the float type.

- **snorm float** - IEEE 32-bit signed-normalized float in range -1 to 1 inclusive.
- **unorm float** - IEEE 32-bit unsigned-normalized float in range 0 to 1 inclusive.

For example, here is a 4-component signed-normalized float-variable declaration.

```
snorm float4 fourComponentIEEEFloat;
```

String Type

HLSL also supports a **string** type, which is an ASCII string. There are no operations or states that accept strings, but effects can query string parameters and annotations.

Example

```
C

// top-level variable
float globalShaderVariable;

// top-level function
void function(
    in float4 position: POSITION0 // top-level argument
)
{
    float localShaderVariable; // local variable
    function2(...)
}

void function2()
{
    ...
}
```

See also

- [Declaring Scalar Types](#)
- [Data Types \(DirectX HLSL\)](#)

Vector Type

Article • 03/04/2021 • 2 minutes to read

A vector contains between one and four scalar components; every component of a vector must be of the same type.

TypeNumber Name

TypeComponents Name

Components

Item	Description
TypeComponents	A single name that contains two parts. The first part is one of the scalar types. The second part is the number of components, which must be between 1 and 4 inclusive.
Name	An ASCII string that uniquely identifies the variable name.

Examples

Here are some examples:

```
bool    bVector;    // scalar containing 1 Boolean
int1    iVector = 1;
float3  fVector = { 0.2f, 0.3f, 0.4f };
```

A vector can be declared using this syntax also:

```
vector <Type, Number> VariableName
```

Here are some examples:

```
vector <int,    1> iVector = 1;
vector <double, 4> dVector = { 0.2, 0.3, 0.4, 0.5 };
```

See also

[Data Types \(DirectX HLSL\)](#)

Matrix Type

Article • 10/24/2019 • 2 minutes to read

A matrix is a special data type that contains between one and sixteen components. Every component of a matrix must be of the same type.

TypeComponents	Name
----------------	------

Components

Item	Description
TypeComponents	A single name that contains three parts. The first part is one of the scalar types. The second part is the number of rows. The third part is the number of columns. The number of rows and columns is a positive integer between 1 and 4 inclusive.
Name	An ASCII string that uniquely identifies the variable name.

Examples

Here are some examples:

```
int1x1    iMatrix;    // integer matrix with 1 row,  1 column
int4x1    iMatrix;    // integer matrix with 4 rows, 1 column
int1x4    iMatrix;    // integer matrix with 1 row, 4 columns
double3x3 dMatrix;   // double matrix with 3 rows, 3 columns

float2x2 fMatrix = { 0.0f, 0.1, // row 1
                     2.1f, 2.2f // row 2
                   };
```

A matrix can be declared using this syntax also:

```
matrix <Type, Number> VariableName
```

The matrix type uses the angle brackets to specify the type, the number of rows, and the number of columns. This example creates a floating-point matrix, with two rows and two

columns. Any of the scalar data types can be used.

Here is an example:

```
matrix <float, 2, 2> fMatrix = { 0.0f, 0.1, // row 1
                                    2.1f, 2.2f // row 2
                                };
```

See also

[Data Types \(DirectX HLSL\)](#)

Per-Component Math Operations

Article • 07/09/2021 • 10 minutes to read

With HLSL, you can program shaders at an algorithm level. To understand the language, you will need to know how to declare variables and functions, use intrinsic functions, define custom data types and use semantics to connect shader arguments to other shaders and to the pipeline.

Once you learn how to author shaders in HLSL, you will need to learn about API calls so that you can: compile a shader for particular hardware, initialize shader constants, and initialize other pipeline state if necessary.

- [The Vector Type](#)
- [The Matrix Type](#)
 - [Matrix Ordering](#)
- [Examples](#)
- [Related topics](#)

The Vector Type

A vector is a data structure that contains between one and four components.

```
bool    bVector;    // scalar containing 1 Boolean
bool1   bVector;    // vector containing 1 Boolean
int1    iVector;    // vector containing 1 int
float3  fVector;    // vector containing 3 floats
double4 dVector;    // vector containing 4 doubles
```

The integer immediately following the data type is the number of components on the vector.

Initializers can also be included in the declarations.

```
bool    bVector = false;
int1    iVector = 1;
float3  fVector = { 0.2f, 0.3f, 0.4f };
double4 dVector = { 0.2, 0.3, 0.4, 0.5 };
```

Alternatively, the vector type can be used to make the same declarations:

```
vector <bool, 1> bVector = false;
vector <int, 1> iVector = 1;
vector <float, 3> fVector = { 0.2f, 0.3f, 0.4f };
vector <double, 4> dVector = { 0.2, 0.3, 0.4, 0.5 };
```

The vector type uses angle brackets to specify the type and number of components.

Vectors contain up to four components, each of which can be accessed using one of two naming sets:

- The position set: x,y,z,w
- The color set: r,g,b,a

These statements both return the value in the third component.

```
// Given
float4 pos = float4(0,0,2,1);

pos.z    // value is 2
pos.b    // value is 2
```

Naming sets can use one or more components, but they cannot be mixed.

```
// Given
float4 pos = float4(0,0,2,1);
float2 temp;

temp = pos.xy // valid
temp = pos.rg // valid

temp = pos.xg // NOT VALID because the position and color sets were used.
```

Specifying one or more vector components when reading components is called swizzling. For example:

```
float4 pos = float4(0,0,2,1);
float2 f_2D;
f_2D = pos.xy;    // read two components
f_2D = pos.xz;    // read components in any order
f_2D = pos.zx;
```

```
f_2D = pos.xx; // components can be read more than once  
f_2D = pos.yy;
```

Masking controls how many components are written.

```
float4 pos = float4(0,0,2,1);  
float4 f_4D;  
f_4D = pos; // write four components  
  
f_4D.xz = pos.xz; // write two components  
f_4D.zx = pos.xz; // change the write order  
  
f_4D.xzyw = pos.w; // write one component to more than one component  
f_4D.wzyx = pos;
```

Assignments cannot be written to the same component more than once. So the left side of this statement is invalid:

```
f_4D.xx = pos.xy; // cannot write to the same destination components
```

Also, the component name spaces cannot be mixed. This is an invalid component write:

```
f_4D.xg = pos.rgrg; // invalid write: cannot mix component name spaces
```

Accessing a vector as a scalar will access the first component of the vector. The following two statements are equivalent.

```
f_4D.a = pos * 5.0f;  
f_4D.a = pos.r * 5.0f;
```

The Matrix Type

A matrix is a data structure that contains rows and columns of data. The data can be any of the scalar data types, however, every element of a matrix is the same data type. The

number of rows and columns is specified with the row-by-column string that is appended to the data type.

```
int1x1    iMatrix;    // integer matrix with 1 row,  1 column
int2x1    iMatrix;    // integer matrix with 2 rows, 1 column
...
int4x1    iMatrix;    // integer matrix with 4 rows, 1 column
...
int1x4    iMatrix;    // integer matrix with 1 row, 4 columns
double1x1 dMatrix;   // double matrix with 1 row,  1 column
double2x2 dMatrix;   // double matrix with 2 rows, 2 columns
double3x3 dMatrix;   // double matrix with 3 rows, 3 columns
double4x4 dMatrix;   // double matrix with 4 rows, 4 columns
```

The maximum number of rows or columns is 4; the minimum number is 1.

A matrix can be initialized when it is declared:

```
float2x2 fMatrix = { 0.0f, 0.1, // row 1
                     2.1f, 2.2f // row 2
                   };
```

Or, the matrix type can be used to make the same declarations:

```
matrix <float, 2, 2> fMatrix = { 0.0f, 0.1, // row 1
                                   2.1f, 2.2f // row 2
                                 };
```

The matrix type uses the angle brackets to specify the type, the number of rows, and the number of columns. This example creates a floating-point matrix, with two rows and two columns. Any of the scalar data types can be used.

This declaration defines a matrix of float values (32-bit floating-point numbers) with two rows and three columns:

```
matrix <float, 2, 3> fFloatMatrix;
```

A matrix contains values organized in rows and columns, which can be accessed using the structure operator `".` followed by one of two naming sets:

- The zero-based row-column position:
 - `_m00, _m01, _m02, _m03`
 - `_m10, _m11, _m12, _m13`
 - `_m20, _m21, _m22, _m23`
 - `_m30, _m31, _m32, _m33`
- The one-based row-column position:
 - `_11, _12, _13, _14`
 - `_21, _22, _23, _24`
 - `_31, _32, _33, _34`
 - `_41, _42, _43, _44`

Each naming set starts with an underscore followed by the row number and the column number. The zero-based convention also includes the letter "m" before the row and column number. Here's an example that uses the two naming sets to access a matrix:

```
// given
float2x2 fMatrix = { 1.0f, 1.1f, // row 1
                      2.0f, 2.1f // row 2
                    };

float f_1D;
f_1D = matrix._m00; // read the value in row 1, column 1: 1.0
f_1D = matrix._m11; // read the value in row 2, column 2: 2.1

f_1D = matrix._11; // read the value in row 1, column 1: 1.0
f_1D = matrix._22; // read the value in row 2, column 2: 2.1
```

Just like vectors, naming sets can use one or more components from either naming set.

```
// Given
float2x2 fMatrix = { 1.0f, 1.1f, // row 1
                      2.0f, 2.1f // row 2
                    };

float2 temp;

temp = fMatrix._m00_m11 // valid
temp = fMatrix._m11_m00 // valid
temp = fMatrix._11_22 // valid
temp = fMatrix._22_11 // valid
```

A matrix can also be accessed using array access notation, which is a zero-based set of indices. Each index is inside of square brackets. A 4x4 matrix is accessed with the following indices:

- [0][0], [0][1], [0][2], [0][3]
- [1][0], [1][1], [1][2], [1][3]
- [2][0], [2][1], [2][2], [2][3]
- [3][0], [3][1], [3][2], [3][3]

Here is an example of accessing a matrix:

```
float2x2 fMatrix = { 1.0f, 1.1f, // row 1
                     2.0f, 2.1f // row 2
                   };
float temp;

temp = fMatrix[0][0] // single component read
temp = fMatrix[0][1] // single component read
```

Notice that the structure operator `"."` is not used to access an array. Array access notation cannot use swizzling to read more than one component.

```
float2 temp;
temp = fMatrix[0][0]_[0][1] // invalid, cannot read two components
```

However, array accessing can read a multi-component vector.

```
float2 temp;
float2x2 fMatrix;
temp = fMatrix[0] // read the first row
```

As with vectors, reading more than one matrix component is called swizzling. More than one component can be assigned, assuming only one name space is used. These are all valid assignments:

```
// Given these variables
float4x4 worldMatrix = float4( {0,0,0,0}, {1,1,1,1}, {2,2,2,2}, {3,3,3,3} );
float4x4 tempMatrix;
```

```
tempMatrix._m00_m11 = worldMatrix._m00_m11; // multiple components
tempMatrix._m00_m11 = worldMatrix.m13_m23;

tempMatrix._11_22_33 = worldMatrix._11_22_33; // any order on swizzles
tempMatrix._11_22_33 = worldMatrix._24_23_22;
```

Masking controls how many components are written.

```
// Given
float4x4 worldMatrix = float4( {0,0,0,0}, {1,1,1,1}, {2,2,2,2}, {3,3,3,3} );
float4x4 tempMatrix;

tempMatrix._m00_m11 = worldMatrix._m00_m11; // write two components
tempMatrix._m23_m00 = worldMatrix._m00_m11;
```

Assignments cannot be written to the same component more than once. So the left side of this statement is invalid:

```
// cannot write to the same component more than once
tempMatrix._m00_m00 = worldMatrix._m00_m11;
```

Also, the component name spaces cannot be mixed. This is an invalid component write:

```
// Invalid use of same component on left side
tempMatrix._11_m23 = worldMatrix._11_22;
```

Matrix Ordering

Matrix packing order for uniform parameters is set to column-major by default. This means each column of the matrix is stored in a single constant register. On the other hand, a row-major matrix packs each row of the matrix in a single constant register. Matrix packing can be changed with the `#pragmapack_matrix` directive, or with the `row_major` or the `column_major` keyword.

The data in a matrix is loaded into shader constant registers before a shader runs. There are two choices for how the matrix data is read: in row-major order or in column-major order. Column-major order means that each matrix column will be stored in a single constant register, and row-major order means that each row of the matrix will be stored

in a single constant register. This is an important consideration for how many constant registers are used for a matrix.

A row-major matrix is laid out like the following:

11

21

31

41

12

22

32

42

13

23

33

43

14

24

34

44

A column-major matrix is laid out like the following:

11

12

13

14

21

22

23

24

31

32

33

34

41
42
43
44

Row-major and column-major matrix ordering determine the order the matrix components are read from shader inputs. Once the data is written into constant registers, matrix order has no effect on how the data is used or accessed from within shader code. Also, matrices declared in a shader body do not get packed into constant registers. Row-major and column-major packing order has no influence on the packing order of constructors (which always follows row-major ordering).

The order of the data in a matrix can be declared at compile time or the compiler will order the data at runtime for the most efficient use.

Examples

HLSL uses two special types, a vector type and a matrix type to make programming 2D and 3D graphics easier. Each of these types contain more than one component; a vector contains up to four components, and a matrix contains up to 16 components. When vectors and matrices are used in standard HLSL equations, the math performed is designed to work per-component. For instance, HLSL implements this multiply:

```
float4 v = a*b;
```

as a four-component multiply. The result is four scalars:

```
float4 v = a*b;  
  
v.x = a.x*b.x;  
v.y = a.y*b.y;  
v.z = a.z*b.z;  
v.w = a.w*b.w;
```

This is four multiplications where each result is stored in a separate component of v. This is called a four-component multiply. HLSL uses component math which makes writing shaders very efficient.

This is very different from a multiply which is typically implemented as a dot product which generates a single scalar:

```
v = a.x*b.x + a.y*b.y + a.z*b.z + a.w*b.w;
```

A matrix also uses per-component operations in HLSL:

```
float3x3 mat1,mat2;  
...  
float3x3 mat3 = mat1*mat2;
```

The result is a per-component multiply of the two matrices (as opposed to a standard 3x3 matrix multiply). A per component matrix multiply yields this first term:

```
mat3.m00 = mat1.m00 * mat2._m00;
```

This is different from a 3x3 matrix multiply which would yield this first term:

```
// First component of a four-component matrix multiply  
mat.m00 = mat1._m00 * mat2._m00 +  
          mat1._m01 * mat2._m10 +  
          mat1._m02 * mat2._m20 +  
          mat1._m03 * mat2._m30;
```

Overloaded versions of the multiply intrinsic function handle cases where one operand is a vector and the other operand is a matrix. Such as: vector * vector, vector * matrix, matrix * vector, and matrix * matrix. For instance:

```
float4x3 World;  
  
float4 main(float4 pos : SV_POSITION) : SV_POSITION  
{  
    float4 val;  
    val.xyz = mul(pos,World);  
    val.w = 0;
```

```
    return val;  
}
```

produces the same result as:

```
float4x3 World;  
  
float4 main(float4 pos : SV_POSITION) : SV_POSITION  
{  
    float4 val;  
    val.xyz = (float3) mul((float1x4)pos,World);  
    val.w = 0;  
  
    return val;  
}
```

This example casts the pos vector to a column vector using the (float1x4) cast. Changing a vector by casting, or swapping the order of the arguments supplied to multiply is equivalent to transposing the matrix.

Automatic cast conversion causes the multiply and dot intrinsic functions to return the same results as used here:

```
{  
    float4 val;  
    return mul(val,val);  
}
```

This result of the multiply is a $1 \times 4 * 4 \times 1 = 1 \times 1$ vector. This is equivalent to a dot product:

```
{  
    float4 val;  
    return dot(val,val);  
}
```

which returns a single scalar value.

Related topics

[Data Types \(DirectX HLSL\)](#)

Sampler Type

Article • 08/25/2021 • 2 minutes to read

Use the following syntax to declare sampler state as well as sampler-comparison state.

Differences between Direct3D 9 and Direct3D 10 and later:

Here is the syntax for a sampler in Direct3D 9.

```
sampler Name = SamplerType{ Texture = <texture_variable>; [state_name = state_value;] ... };
```

The syntax for a sampler in Direct3D 10 and later is changed slightly to support texture objects and sampler arrays.

```
SamplerType Name[Index]{ [state_name = state_value;] ... };
```

Parameters

sampler

Direct3D 9 only. Required keyword.

Name

ASCII string that uniquely identifies the sampler variable name.

[Index]

Direct3D 10 and later only. Optional array size; a positive integer greater than or equal to 1.

SamplerType

[in] The sampler type, which is one of the following: *sampler*, *sampler1D*, *sampler2D*, *sampler3D*, *samplerCUBE*, *sampler_state*, *SamplerState*.

Differences between Direct3D 9 and Direct3D 10 and later:

- Direct3D 10 and later supports one additional sampler type:
SamplerComparisonState.

Texture = <*texture_variable*>;

Direct3D 9 only. A texture variable. The *Texture* keyword is required on the left-hand side; the variable name belongs on the right-hand side of the expression within the angle brackets.

state_name = *state_value*

[in] Optional state assignment(s). The left hand side of an assignment is a state name, the right hand side is the state value. All state assignments must appear within a statement block (in curly brackets). Each statement is separated with a semicolon. The following table lists the possible state names.

```
// sampler state
AddressU
AddressV
AddressW
BorderColor
Filter
MaxAnisotropy
MaxLOD
MinLOD
MipLODBias

// sampler-comparison state
ComparisonFunc
```

The right side of each expression is the value assigned to each state. See the [D3D11_SAMPLER_DESC](#) structure for the possible state values for Direct3D 11. There is a 1 to 1 relationship between the state names and the members of the structure. See the following example.

Remarks

When you implement an effect, sampler state is one of several types of state that you might need to set up in the pipeline for rendering. For a list of all the possible states that you can set in an effect, see:

- Direct3D 10 uses [state groups](#).
- Direct3D 9 uses individual [states](#).

Example

Differences between Direct3D 9 and Direct3D 10:

Here is a partial example of a Direct3D 9 sampler from [BasicHLSL Sample](#).

```
    sampler MeshTextureSampler =
    sampler_state
    {
        Texture = <g_MeshTexture>;
        MipFilter = LINEAR;
        MinFilter = LINEAR;
        MagFilter = LINEAR;
    };
```

Here is a partial example of a Direct3D 10 sampler from [BasicHLSL10 Sample](#).

```
SamplerState MeshTextureSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = Wrap;
    AddressV = Wrap;
};
```

Here is a partial example of declaring sampler-comparison state, and calling a comparison sampler in Direct3D 10.

```
SamplerComparisonState ShadowSampler
{
    // sampler state
    Filter = COMPARISON_MIN_MAG_LINEAR_MIP_POINT;
    AddressU = MIRROR;
    AddressV = MIRROR;

    // sampler comparison state
    ComparisonFunc = LESS;
};

float3 vModProjUV;
...
float fShadow = g_ShadowMap.SampleCmpLevelZero( ShadowSampler,
vModProjUV.xy, vModProjUV.z);
```

See also

Data Types (DirectX HLSL)

Shader Type

Article • 08/19/2020 • 2 minutes to read

The syntax for declaring a shader variable in an effect changed from Direct3D 9 to Direct3D 10.

Shader Type for Direct3D 10

Declare a shader variable within an effect pass (in Direct3D 10) using the shader type syntax:

```
SetPixelShader Compile( ShaderTarget, ShaderFunction ); SetGeometryShader Compile( ShaderTarget, ShaderFunction ); SetVertexShader Compile( ShaderTarget, ShaderFunction );
```

Parameters

Item	Description
SetXXXShader	The Direct3D API call that creates the shader object. Can be either: SetPixelShader or SetGeometryShader or SetVertexShader .
ShaderTarget	The shader model to compile against. This is valid for any target including all Direct3D 9 targets plus the shader model 4 targets: vs_4_0, gs_4_0, and ps_4_0.
ShaderFunction	An ASCII string that contains the name of the shader entry point function; this is the function that begins execution when the shader is invoked. The (...) represents the shader arguments; these are the same arguments passed to the shader creation API's: VSSetShader or GSSetShader or PSSetShader .

Example

Here is an example that creates a vertex shader and pixel shader object, compiled for a particular shader model. In the Direct3D 10 example, there is no geometry shader, so the pointer is set to **NULL**.

```
// Direct3D 10
technique10 Render
{
    pass P0
{
```

```

        SetVertexShader( CompileShader( vs_4_0, VS() ) );
        SetGeometryShader( NULL );
        SetPixelShader( CompileShader( ps_4_0, PS() ) );
    }
}

```

Shader Type for Direct3D 9

Declare a shader variable within an effect pass (for Direct3D 9) using the shader type syntax:

PixelShader = compile ShaderTarget ShaderFunction (...); VertexShader = compile ShaderTarget ShaderFunction (...);

Parameters

Item	Description
XXXShader	A shader variable, which represents the compiled shader. Can be either: PixelShader or VertexShader .
ShaderTarget	The shader model to compile against; depends on the type of shader variable.
ShaderFunction (...)	An ASCII string that contains the name of the shader entry point function; this is the function that begins execution when the shader is invoked. The (...) represents the shader arguments; these are the same arguments passed to the shader creation API's: SetVertexShader or SetPixelShader .

Example

Here is an example of a vertex shader and pixel shader object, compiled for a particular shader model.

```

// Direct3D 9
technique RenderSceneWithTexture1Light
{
    pass P0
    {
        VertexShader = compile vs_2_0 RenderSceneVS( 1, true, true );
        PixelShader = compile ps_2_0 RenderScenePS( true );
    }
}

```

See also

[Data Types \(DirectX HLSL\)](#)

Texture type

Article • 05/07/2020 • 2 minutes to read

Use the following syntax to declare a texture variable.

Type Name:

Parameters

Item	Description
Type	One of the following types: texture (untyped, for backwards compatibility), Texture1D, Texture1DArray, Texture2D, Texture2DArray, Texture3D, TextureCube. The element size must fit into 4 32-bit quantities.
Name	An ASCII string that uniquely identifies the variable name.

Remarks

There are three parts to using a texture.

1. Declaring a texture variable. This is done with the syntax shown above. For example, these are valid declarations.

```
texture g_MeshTexture;
```

- or -

```
Texture2D g_MeshTexture;
```

2. Declaring and initializing a sampler object. This is done with slightly different syntax in Direct3D 9 and Direct3D 10. For details about sampler object syntax, see [Sampler Type \(DirectX HLSL\)](#).
3. Invoking a texture function in a shader.

Differences between Direct3D 9 and Direct3D 10:

Direct3D 9 uses [intrinsic texture functions](#) to perform texture operations. This example is from the [BasicHLSL Sample](#) and uses [tex2D\(s, t\) \(DirectX HLSL\)](#) to perform texture sampling.

```
Output.RGBColor = tex2D(MeshTextureSampler, In.TextureUV) * In.Diffuse;
```

Direct3D 10 uses [templated texture objects](#) instead. Here is an example of the equivalent texture operation.

```
Output.RGBColor = g_MeshTexture.Sample(MeshTextureSampler, In.TextureUV) *  
In.Diffuse;
```

See also

[Data Types \(DirectX HLSL\)](#)

Struct Type

Article • 06/30/2021 • 2 minutes to read

Use the following syntax to declare a structure using HLSL.

```
struct Name{ [InterpolationModifier] Type[RxC] MemberName; ... };
```

Parameters

Name

An ASCII string that uniquely identifies the structure name.

[*InterpolationModifier*]

Optional modifier that specifies an interpolation type. See [Remarks](#) for details.

Type[RxC]

The member type with an optional row (*R*) x column (*C*) array size. A structure contains at least one element; if it contains more than one element, the elements are all of the same type. The number of rows and columns are unsigned integers between 1 and 4 inclusive.

MemberName

An ASCII string that uniquely identifies the member name.

Remarks

An interpolation modifier can be specified on any structure member or on an argument to a pixel shader function. If a modifier appears in both places, the outside modifier (the pixel shader argument modifier) overrules the inside modifier (the structure modifier).

When compiling a shader or an effect, the shader compiler packs structure members according to [HLSL packing rules](#).

Interpolation Modifiers Introduced in Shader Model 4

Vertex shader outputs that are used for pixel shader inputs are linearly interpolated to get per-pixel values during rasterization. To set the method of interpolation, use any of

the following values, which are supported in [shader model 4](#) or later. The modifier is ignored on any vertex shader output that is not used as a pixel shader input.

Interpolation Modifier	Description
linear	Interpolate between shader inputs; linear is the default value if no interpolation modifier is specified.
centroid	Interpolate between samples that are somewhere within the covered area of the pixel (this may require extrapolating end points from a pixel center). Centroid sampling may improve antialiasing if a pixel is partially covered (even if the pixel center is not covered). The centroid modifier must be combined with either the linear or noperspective modifier.
nointerp	Do not interpolate .
noperspective	Do not perform perspective-correction during interpolation. The noperspective modifier can be combined with the centroid modifier.
sample	Available in shader model 4.1 and later Interpolate at sample location rather than at the pixel center. This causes the pixel shader to execute per-sample rather than per-pixel. Another way to cause per-sample execution is to have an input with semantic SV_SampleIndex , which indicates the current sample. Only the inputs with sample specified (or inputting SV_SampleIndex) differ between shader invocations in the pixel, whereas other inputs that do not specify modifiers (for example, if you mix modifiers on different inputs) still interpolate at the pixel center. Both pixel shader invocation and depth/stencil testing occur for every covered sample in the pixel. This is sometimes known as <i>supersampling</i> . In contrast, in the absence of sample frequency invocation, known as <i>multisampling</i> , the pixel shader is invoked once per pixel regardless of how many samples are covered, while depth/stencil testing occurs at sample frequency. Both modes provide equivalent edge antialiasing. However, supersampling provides better shading quality by invoking the pixel shader more frequently.

1. When using an int/uint type, the only valid option is ****nointerp****.

Interpolation modifiers can be applied to structure members or [function arguments](#), or both.

Examples

Here are some example structure declarations.

```
struct struct1
{
```

```
    int    a;  
}
```

This declaration includes an array.

```
struct struct2  
{  
    int    a;  
    float   b;  
    int4x4  iMatrix;  
}
```

This declaration includes an interpolation modifier.

```
struct In  
{  
    centroid float2 Texcoord;  
};
```

See also

[Data Types \(DirectX HLSL\)](#)

User-Defined Type

Article • 08/23/2019 • 2 minutes to read

Use the following syntax to declare a user-defined type.

```
typedef [const] Type Name[Index];
```

Parameters

Item	Description
[const]	Optional. This keyword explicitly marks the type as a constant.
Type	Identifies the data type; must be one of the HLSL intrinsic data types.
Name	An ASCII string that uniquely identifies the variable name.
Index	Optional array size. Must be an unsigned integer between 1 and 4 inclusive.

In addition to the built-in intrinsic data types, HLSL supports user-defined or custom types which follow this syntax:

Remarks

User-defined types are not case-sensitive. For convenience, the following types are automatically defined at super-global scope.

```
typedef vector <bool, #> bool#;
typedef vector <int, #> int#;
typedef vector <uint, #> uint#;
typedef vector <half, #> half#;
typedef vector <float, #> float#;
typedef vector <double, #> double#;

typedef matrix <bool, #, #> bool#x#;
typedef matrix <int, #, #> int#x#;
typedef matrix <uint, #, #> uint#x#;
typedef matrix <half, #, #> half#x#;
typedef matrix <float, #, #> float#x#;
typedef matrix <double, #, #> double#x#;
```

The pound sign (#) represents an integer digit between 1 and 4.

For compatibility with DirectX 8 effects, the following types are automatically defined at super-global scope:

```
typedef int DWORD;
typedef float FLOAT;
typedef vector <float, 4> VECTOR;
typedef matrix <float, 4, 4> MATRIX;
typedef string STRING;
typedef texture TEXTURE;
typedef pixelshader PIXELSHADER;
typedef vertexshader VERTEXSHADER;
```

Related topics

[Data Types \(DirectX HLSL\)](#)

State Objects

Article • 03/05/2021 • 5 minutes to read

With shader models 6.3 and later, applications have the convenience and flexibility of being able to define DXR state objects directly in HLSL shader code in addition to using Direct3D 12 APIs.

In HLSL, state objects are declared with this syntax:

syntax

```
Type Name =  
{  
    Field1,  
    Field2,  
    ...  
};
```

Item	Description
Type	Identifies the type of subobject. Must be one of the supported HLSL subobject types.
Name	An ASCII string that uniquely identifies the variable name.
Field[1, 2, ...]	Fields of the subobject. Specific fields for each type of subobject are described below.

List of subobject types:

- [StateObjectConfig](#)
- [GlobalRootSignature](#)
- [LocalRootSignature](#)
- [SubobjectToExportsAssociation](#)
- [RaytracingShaderConfig](#)
- [RaytracingPipelineConfig](#)
- [TriangleHitGroup](#)
- [ProceduralPrimitiveHitGroup](#)

StateObjectConfig

The StateObjectConfig subobject type corresponds to a [D3D12_STATE_OBJECT_CONFIG](#) structure.

It has one field, a bitwise flag, which is one or both of

- STATE_OBJECT_FLAGS_ALLOW_LOCAL_DEPENDENCIES_ON_EXTERNAL_DEFINITONS
- STATE_OBJECT_FLAGS_ALLOW_EXTERNAL_DEPENDENCIES_ON_LOCAL_DEFINITIONS

or, zero for neither of them.

Example:

```
StateObjectConfig MyStateObjectConfig =
{
    STATE_OBJECT_FLAGS_ALLOW_LOCAL_DEPENDENCIES_ON_EXTERNAL_DEFINITONS
};
```

GlobalRootSignature

A GlobalRootSignature corresponds to a [D3D12_GLOBAL_ROOT_SIGNATURE](#) structure.

The fields consist of some number of strings describing the parts of the root signature. For reference on this, see [Specifying Root Signatures in HLSL](#).

Example:

```
GlobalRootSignature MyGlobalRootSignature =
{
    "DescriptorTable(UAV(u0)),"
        // Output texture
    "SRV(t0),"
        // Acceleration
    structure
        "CBV(b0),"
            // Scene constants
        "DescriptorTable(SRV(t1, numDescriptors = 2))" // Static index and
    vertex buffers.
};
```

LocalRootSignature

A LocalRootSignature corresponds to a [D3D12_LOCAL_ROOT_SIGNATURE](#) structure.

Just like the global root signature subobject, the fields consist of some number of strings describing the parts of the root signature. For reference on this, see [Specifying Root Signatures in HLSL](#).

Example:

```
LocalRootSignature MyLocalRootSignature =
{
    "RootConstants(num32BitConstants = 4, b1)" // Cube constants
};
```

SubobjectToExportsAssociation

By default, a subobject merely declared in the same library as an export is able to apply to that export. However, applications have the ability to override that and get specific about what subobject goes with which export. In HLSL, this "explicit association" is done using SubobjectToExportsAssociation.

A SubobjectToExportsAssociation corresponds to a [D3D12_DXIL_SUBOBJECT_TO_EXPORTS_ASSOCIATION](#) structure.

This subobject is declared with the syntax

```
syntax

SubobjectToExportsAssociation Name =
{
    SubObjectName,
    Exports
};
```

Item	Description
Name	An ASCII string that uniquely identifies the variable name.
SubObjectName	String which identifies an exported subobject.
Exports	String containing a semicolon-delimited list of exports.

Example:

```
SubobjectToExportsAssociation MyLocalRootSignatureAssociation =
{
    "MyLocalRootSignature", // Subobject name
    "MyHitGroup;MyMissShader" // Exports association
};
```

Note that both fields use *exported* names. An exported name may be different from the original name in HLSL, if the application chooses to do export-renaming.

RaytracingShaderConfig

A RaytracingShaderConfig corresponds to a [D3D12_RAYTRACING_SHADER_CONFIG](#) structure.

This subobject is declared with the syntax

syntax

```
RaytracingShaderConfig Name =
{
    MaxPayloadSize,
    MaxAttributeSize
};
```

Item	Description
Name	An ASCII string that uniquely identifies the variable name.
MaxPayloadSize	Numerical value for the maximum storage for scalars (counted as 4 bytes each) in ray payloads for associated raytracing shaders.
MaxAttributeSize	Numerical value for the maximum number of scalars (counted as 4 bytes each) that can be used for attributes in associated raytracing shaders. The value cannot exceed D3D12_RAYTRACING_MAX_ATTRIBUTE_SIZE_IN_BYTES .

Example:

```
RaytracingShaderConfig MyShaderConfig =
{
    16, // Max payload size
    8   // Max attribute size
};
```

RaytracingPipelineConfig

A RaytracingPipelineConfig corresponds to a [D3D12_RAYTRACING_PIPELINE_CONFIG](#) structure.

This subobject is declared with the syntax

syntax

```
RaytracingPipelineConfig Name =  
{  
    MaxTraceRecursionDepth  
};
```

Item	Description
Name	An ASCII string that uniquely identifies the variable name.
MaxTraceRecursionDepth	Numerical limit to use for ray recursion in the raytracing pipeline. It is a number between 0 and 31, inclusive.

Example:

```
RaytracingPipelineConfig MyPipelineConfig =  
{  
    1 // Max trace recursion depth  
};
```

Since there is a performance cost to raytracing recursion, applications should use the lowest recursion depth needed for the desired results.

If shader invocations haven't yet reached the maximum recursion depth, they can call [TraceRay](#) any number of times. But if they reach or exceed the maximum recursion depth, calling TraceRay puts the device into removed state. Therefore, raytracing shaders should take care to stop calling TraceRay if they've met or exceeded the maximum recursion depth.

TriangleHitGroup

A TriangleHitGroup corresponds to a [D3D12_HIT_GROUP_DESC](#) structure whose Type field is set to [D3D12_HIT_GROUP_TYPE_TRIANGLES](#).

This subobject is declared with the syntax

syntax

```
TriangleHitGroup Name =  
{  
    AnyHitShader,  
    ClosestHitShader  
};
```

Item	Description
Name	An ASCII string that uniquely identifies the variable name.
AnyHitShader	String name of the anyhit shader for the hit group, or an empty string.
ClosestHitShader	String name of the closest hit shader for the hit group, or an empty string.

Example:

```
TriangleHitGroup MyHitGroup =
{
    "",           // AnyHit
    "MyClosestHitShader", // ClosestHit
};
```

Note that both fields use *exported* names. An exported name may be different from the original name in HLSL, if the application chooses to do export-renaming.

ProceduralPrimitiveHitGroup

A ProceduralPrimitiveHitGroup corresponds to a [D3D12_HIT_GROUP_DESC](#) structure whose Type field is set to [D3D12_HIT_GROUP_TYPE PROCEDURAL_PRIMITIVE](#).

This subobject is declared with the syntax

syntax

```
ProceduralPrimitiveHitGroup Name =
{
    AnyHitShader,
    ClosestHitShader,
    IntersectionShader
};
```

Item	Description
Name	An ASCII string that uniquely identifies the variable name.
AnyHitShader	String name of the anyhit shader for the hit group, or an empty string.
ClosestHitShader	String name of the closest hit shader for the hit group, or an empty string.
IntersectionShader	String name of the intersection shader for the hit group, or an empty string.

Example:

```
ProceduralPrimitiveHitGroup MyProceduralHitGroup
{
    "MyAnyHit",      // AnyHit
    "MyClosestHit", // ClosestHit
    "MyIntersection" // Intersection
};
```

Note that the three fields use *exported* names. An exported name may be different from the original name in HLSL, if the application chooses to do export-renaming.

Remarks

Subobjects have the notion of "association", or "which subobject goes with which export".

When specifying subobjects through shader code, the choice of "which subobject goes with which export" follows the rules as outlined in the [DXR specification](#). In particular, suppose an application has some export. If an application associates that export with root signature A through shader-code and root signature B through application code, B is the one that gets used. The design of "use B" instead of "produce an error" gives applications the ability to conveniently override DXIL associations using application code, rather than be forced to recompile shaders to resolve mismatching things.

Related topics

[DirectX Developer Blog post "New in D3D12 – DirectX Raytracing \(DXR\) now supports library subobjects"](#)

[DirectX Raytracing \(DXR\) Functional Spec](#)

[Sample: D3D12RaytracingLibrarySubobjects](#)

Flow Control

Article • 08/23/2019 • 2 minutes to read

Most hardware is designed to run shader code line by line, executing each HLSL statement once. A flow-control statement determines at run time which block of HLSL statements to execute next. Using a flow-control statement, a shader can loop through a set of statements, or jump (branch) to an instruction other than the one on the next line. Some flow-control statements support static control that is specified at compile time; others offer predicated control which is a per-component decision made at runtime, and still others support dynamic control which is a decision made at run time based on the contents of a variable.

HLSL supports the following flow-control statements.

- [break](#)
- [continue](#)
- [discard](#)
- [do](#)
- [for](#)
- [if](#)
- [switch](#)
- [while](#)

Related topics

[Language Syntax \(DirectX HLSL\)](#)

break Statement

Article • 06/30/2021 • 2 minutes to read

Exit the surrounding loop ([do](#), [for](#), [while](#)).

```
break;
```

Parameters

None

See also

[Flow Control](#)

continue Statement

Article • 06/30/2021 • 2 minutes to read

Stop executing the current loop ([do](#), [for](#), [while](#)), update the loop conditions, and begin executing from the top of the loop.

`continue;`

Parameters

None

See also

[Flow Control](#)

discard Statement

Article • 06/30/2021 • 2 minutes to read

Do not output the result of the current pixel.

```
discard;
```

Parameters

None

Remarks

This statement can only be called from a pixel shader; it is not supported within a geometry shader or a vertex shader.

See also

[Flow Control](#)

do Statement

Article • 06/30/2021 • 2 minutes to read

Execute a series of statements continuously until the conditional expression fails.

[*Attribute*] do { *Statement Block*; } while(*Conditional*);

Parameters

Attribute

An optional parameter that controls how the statement is compiled.

Attribute	Description
fastopt	Reduces the compile time but produces less aggressive optimizations. If you use this attribute, the compiler will not unroll loops. This attribute affects only shader model targets that support <code>break</code> instructions. This attribute is available in shader model <code>vs_2_x</code> and <code>shader model 3</code> and later. It is particularly useful in <code>shader model 4</code> and later when the compiler compiles loops. The compiler simulates loops by default to evaluate whether it can unroll them. If you do not want the compiler to unroll loops, use this attribute to reduce compile time.

Statement Block

One or more [statements](#).

Conditional

A conditional [expression](#). The statement block is executed before the expression is evaluated. The loop is exited when the expression evaluates to false.

Requirements

Requirement	Value
Header	Ocidl.h

See also

[Flow Control](#)

for Statement

Article • 06/30/2021 • 2 minutes to read

Iteratively executes a series of statements, based on the evaluation of the conditional expression.

[*Attribute*] for (*Initializer*; *Conditional*; *Iterator*) { *Statement Block*; }

Parameters

Attribute

An optional parameter that controls how the statement is compiled. When no attribute is specified the compiler will first attempt to emit a rolled version of the loop, and if that fails, or if some operations would be easier if the loop was unrolled, will fall back to an unrolled version of the loop.

Attribute	Description
unroll(x)	Unroll the loop until it stops executing. Can optionally specify the maximum number of times the loop is to execute. Not compatible with the [loop] attribute.
loop	Generate code that uses flow control to execute each iteration of the loop. Not compatible with the [unroll] attribute.
fastopt	Reduces the compile time but produces less aggressive optimizations. If you use this attribute, the compiler will not unroll loops. This attribute affects only shader model targets that support break instructions. This attribute is available in shader model vs_2_x and shader model 3 and later. It is particularly useful in shader model 4 and later when the compiler compiles loops. The compiler simulates loops by default to evaluate whether it can unroll them. If you do not want the compiler to unroll loops, use this attribute to reduce compile time.
allow_uav_condition	Allows a compute shader loop termination condition to be based off of a UAV read. The loop must not contain synchronization intrinsics.

Initializer

The initial value of the loop counter.

Conditional

A conditional [expression](#). If the conditional expression evaluates to true, the statement block is executed. The loop ends when the expression evaluates to false.

Iterator

Update the value of the loop counter.

Statement Block

One or more [HLSL statements](#).

Remarks

The `[unroll]` and `[loop]` attributes are mutually exclusive and will generate compiler errors when both are specified.

The `[fastopt]` and `[allow_uav_condition]` attributes are ignored if `[unroll]` is specified.

See also

[Flow Control](#)

if Statement

Article • 08/19/2021 • 2 minutes to read

Conditionally execute a series of statements, based on the evaluation of the conditional expression.

[*Attribute*] if (*Conditional*) { *Statement Block*; }

Parameters

Attribute

An optional parameter that controls how the statement is compiled.

Attribute	Description
branch	Evaluate only one side of the if statement depending on the given condition. [!Note] When you use Shader Model 2.x or Shader Model 3.0 , each time you use dynamic branching you consume resources. So, if you use dynamic branching excessively when you target these profiles, you can receive compilation errors.
flatten	Evaluate both sides of the if statement and choose between the two resulting values.

Conditional

A conditional [expression](#). The expression is evaluated, and if true, the statement block is executed.

Statement Block

One or more [HLSL statements](#).

Remarks

When the compiler uses the branch method for compiling an if statement it will generate code that will evaluate only one side of the if statement depending on the given condition. For example, in the if statement:

```
[branch] if(x)
{
    x = sqrt(x);
}
```

The **if** statement has an implicit **else** block, which is equivalent to $x = x$. Because we have told the compiler to use the **branch** method with the preceding **branch** attribute, the compiled code will evaluate x and execute only the side that should be executed; if x is zero, then it will execute the **else** side, and if it is non-zero it will execute the **then** side.

Conversely, if the **flatten** attribute is used, then the compiled code will evaluate both sides of the **if** statement and choose between the two resulting values using the original value of x . Here is an example of a usage of the **flatten** attribute:

```
[flatten] if(x)
{
    x = sqrt(x);
}
```

There are certain cases where using the **branch** or **flatten** attributes may generate a compile error. The **branch** attribute may fail if either side of the **if** statement contains a gradient function, such as **tex2D**. The **flatten** attribute may fail if either side of the **if** statement contains a stream append statement or any other statement that has side-effects.

An **if** statement can also use an optional **else** block. If the **if** expression is true, the code in the statement block associated with the **if** statement is processed. Otherwise, the statement block associated with the optional **else** block is processed.

See also

[Flow Control](#)

switch Statement

Article • 08/19/2021 • 2 minutes to read

Transfer control to a different statement block within the switch body depending on the value of a selector.

```
[Attribute] switch( Selector ) { case 0 : { StatementBlock; } break; case 1 : { StatementBlock; } break; case n : { StatementBlock; } break; default : { StatementBlock; } break;
```

Parameters

Attribute

An optional parameter that controls how the statement is compiled. When no attribute is specified, the compiler may use a hardware switch or emit a series of **if** statements.

Attribute	Description
flatten	Compile the statement as a series of if statements, each with the flatten attribute.
branch	Compile the statement as a series of if statements each with the branch attribute. [!Note] When you use Shader Model 2.x or Shader Model 3.0 , each time you use dynamic branching you consume resources. So, if you use dynamic branching excessively when you target these profiles, you can receive compilation errors.
forcecase	Force a switch statement in the hardware. [!Note] Requires feature level 10_0 or later hardware.
call	The bodies of the individual cases in the switch will be moved into hardware subroutines and the switch will be a series of subroutine calls. [!Note] Requires feature level 10_0 or later hardware.

Selector

A variable. The case statements inside the curly brackets will each check this variable to see if the SwitchValue matches their particular CaseValue.

StatementBlock

One or more [statements](#).

Remarks

```
[branch] switch(a)
{
    case 0:
        return 0;
    case 1:
        return 1;
    case 2:
        return 3;
    default:
        return 6;
}
```

Is equivalent to:

```
[branch] if( a == 2 )
    return 3;
else if( a == 1 )
    return 1;
else if( a == 0 )
    return 0;
else
    return 6;
```

Here are example usages of forcecase and call flow control attributes:

```
[forcecase] switch(a)
{
    case 0:
        return 0;
    case 1:
        return 1;
    case 2:
        return 3;
    default:
```

```
        return 6;
}

[call] switch(a)
{
    case 0:
        return 0;
    case 1:
        return 1;
    case 2:
        return 3;
default:
    return 6;
}
```

Requirements

Requirement	Value
Header	Urlmon.h

See also

[Flow Control](#)

while Statement

Article • 06/08/2021 • 2 minutes to read

Executes a statement block until the conditional expression fails.

[*Attribute*] while (*Conditional*) { *Statement Block*; }

Parameters

Attribute

An optional parameter that controls how the statement is compiled.

Attribute	Description
unroll(x)	Unroll the loop until it stops executing. Optionally, you can specify the maximum number of times the loop can execute.
loop	Use flow-control statements in the compiled shader; do not unroll the loop.
fastopt	Reduces the compile time but produces less aggressive optimizations. If you use this attribute, the compiler will not unroll loops. This attribute affects only shader model targets that support break instructions. This attribute is available in shader model vs_2_x and shader model 3 and later. It is particularly useful in shader model 4 and later when the compiler compiles loops. The compiler simulates loops by default to evaluate whether it can unroll them. If you do not want the compiler to unroll loops, use this attribute to reduce compile time.
allow_uav_condition	Allows a compute shader loop termination condition to be based off of a UAV read. The loop must not contain synchronization intrinsics.

Conditional

A conditional [expression](#). If the expression evaluates to true, the statement block is executed. The loop ends when the expression evaluates to false.

Statement Block

One or more [statements](#).

See also

[Flow Control](#)

Functions (HLSL reference)

Article • 11/23/2019 • 2 minutes to read

Functions encapsulate HLSL statements. This enables you to debug a set of functions and then reuse them across shaders or effects. You may want to create a function that encapsulates the functionality of a vertex shader, pixel shader or texture shader. Other times, you may want to write a helper function that performs some commonly used task, and then call that helper function from your shader function. The rules for writing shader functions for HLSL are very similar to writing C functions.

- [Syntax](#)
- [Parameters](#)
- [Return Statement](#)
- [Signatures](#)

HLSL also has a number of built-in [Intrinsic Functions \(DirectX HLSL\)](#). Because all intrinsic functions are tested and performance optimized, it is good practice to use an intrinsic function where possible instead of creating your own function.

Related topics

[Language Syntax \(DirectX HLSL\)](#)

Function Declaration Syntax

Article • 06/30/2021 • 2 minutes to read

HLSL functions are declared with the following syntax.

```
[StorageClass] [clipplanes()] [precise] Return_Value Name ( [ArgumentList] ) [:Semantic] {  
    [StatementBlock] ;}
```

Parameters

StorageClass

Modifier that redefines a function declaration. **inline** is currently the only modifier value. The modifier value must be **inline** because it is also the default value. Therefore, a function is inline regardless of whether you specify **inline**, and all functions in HLSL are inline. An inline function generates a copy of the function body (when compiling) for each function call. This is done to decrease the overhead of calling the function.

Clipplanes

Optional list of clip planes, which is up to 6 user-specified clip planes. This is an alternate mechanism for [SV_ClipDistance](#) that works on [feature level 9_x](#) and higher.

Name

An ASCII string that uniquely identifies the name of the shader function.

ArgumentList

Optional argument list, which is a comma-separated list of [arguments](#) passed into a function.

Semantic

Optional string that identifies the intended usage of the return data (see [Semantics \(DirectX HLSL\)](#)).

StatementBlock

Optional [statements](#) that make up the body of the function. A function defined without a body is called a function prototype; the body of a prototype function must be defined elsewhere before the function can be called.

Return Value

The return type can be any one of these [HLSL types](#).

Remarks

The syntax on this page describes almost every type of HLSL function, this includes vertex shaders, pixel shaders, and helper functions. While a geometry shader is also implemented with a function, its syntax is a little more complicated, so there is a separate page that defines a geometry shader function declaration (see [Geometry-Shader Object \(DirectX HLSL\)](#)).

A function can be overloaded as long as it is given a unique combination of parameter types and/or parameter order. HLSL also implements a number of built in, or [intrinsic functions](#).

You can specify user-specific clip planes with the `clipplanes` attribute. Windows applies these clip planes to all of the primitives drawn. The `clipplanes` attribute works like [SV_ClipDistance](#) but works on all hardware [feature level](#) 9_x and higher. For more info, see [User clip planes on feature level 9 hardware](#).

Examples

This example is from BasicHLSL10.fx from the [BasicHLSL10 Sample](#).

```
hlsl

struct VS_OUTPUT
{
    float4 Position    : SV_POSITION;
    float4 Diffuse     : COLOR0;
    float2 TextureUV  : TEXCOORD0;
};

VS_OUTPUT RenderSceneVS( float4 vPos : POSITION,
                        float3 vNormal : NORMAL,
                        float2 vTexCoord0 : TEXCOORD,
                        uniform int nNumLights,
                        uniform bool bTexture,
                        uniform bool bAnimate )
{
    VS_OUTPUT Output;
    ...
    return Output;
}
```

This example from AdvancedParticles.fx from the [AdvancedParticles Sample](#), illustrates using a semantic for the return type.

```
hlsl

//  
// PS for particles  
//  
float4 PSSprite(PSSceneIn input) : SV_Target  
{  
    return g_txDiffuse.Sample( g_samLinear, input.tex ) * input.color;  
}
```

Related topics

[Functions \(DirectX HLSL\)](#)

Function Arguments

Article • 08/20/2021 • 2 minutes to read

A function takes one or more input arguments; use the following syntax to declare each argument.

[InputModifier] Type Name [: Semantic] [InterpolationModifier] [= Initializers]

[Modifier] Type Name [: Semantic] [: Interpolation Modifier] [= Initializer(s)]

If there are multiple function arguments, they are separated by commas.

Parameters

Item	Description	
InputModifier	Optional term that identifies an argument as an input, an output, or both.	
	Value	Description
	in	Input only
	inout	Input and an output
	out	Output only
	uniform	Input only constant data
Parameters are always passed by value. in indicates that the value of the parameter should be copied in from the calling application before the function begins. out indicates that the last value of the parameter should be copied out, and returned to the calling application when the function returns. inout is a shorthand for specifying both.		
A uniform value comes from a constant register; each vertex shader or pixel shader invocation see the same initial value for a uniform variable. Global variables are treated as if they are declared uniform. For non-top-level functions, uniform is synonymous with in. If no parameter usage is specified, the parameter usage is assumed to be in.		
Type	The argument type; can be any valid HLSL type .	
Name	An ASCII string that uniquely identifies the name of the shader function.	

Item	Description
Semantic	Optional string that identifies the intended usage of the data (see Semantics (DirectX HLSL)).
InterpolationModifier	Optional interpolation modifier which allows a shader to determine the method of interpolation. An interpolation modifier on a function argument only applies to an argument used as an input to a pixel shader function.
Initializers	Optional values for initialization; multiple values are required to initialize multi-component data types.

Remarks

Function arguments are listed in a comma-separated argument list in a function declaration. As in C functions, each argument must have a parameter name and type declared; an argument to an HLSL function optionally can include a semantic, an initial value, and a pixel shader input can include an interpolation type.

The *Type* of a function argument could be a structure, which could include a per-member interpolation modifier. If the function argument also has an interpolation modifier, the function argument modifier overrides interpolation modifiers declared within the Type.

Examples

This example (from the [BasicHLSL10 Sample](#)) illustrates uniform and non-uniform inputs to a vertex shader function.

```
VS_OUTPUT RenderSceneVS(
    float4 vPos : POSITION,
    float3 vNormal : NORMAL,
    float2 vTexCoord0 : TEXCOORD,
    uniform int nNumLights,
    uniform bool bTexture,
    uniform bool bAnimate )
{
    ...
}
```

This example (from the [ContentStreaming Sample](#)) uses an input structure to pass arguments to a pixel shader function.

```
VSBasicIn input
struct VSBasicIn
{
    float4 Pos      : POSITION;
    float3 Norm     : NORMAL;
    float2 Tex      : TEXCOORD0;
};

PSBasicIn VSBasic(VSBasicIn input)
{
    ...
}
```

Related topics

[Functions \(DirectX HLSL\)](#)

return Statement

Article • 06/30/2021 • 2 minutes to read

A return statement signals the end of a function.

```
return [value];
```

The simplest return statement returns control from the function to the calling program; it returns no value.

```
void main()
{
    return ;
}
```

However, a return statement can return one or more values. This example returns a literal value:

```
float main( float input : COLOR0 ) : COLOR0
{
    return 0;
}
```

This example returns the scalar result of an expression.

```
return light.enabled = true ;
```

This example returns a four-component vector that is constructed from a local variable and a literal.

```
return float4(color.rgb, 1) ;
```

This example returns a four-component vector that is constructed from the result that is returned from an intrinsic function, together with literal values.

```
float4 func(float2 a: POSITION): COLOR
{
    return float4(sin(length(a) * 100.0) * 0.5 + 0.5, sin(a.y * 50.0), 0,
1);
}
```

This example returns a structure that contains one or more members.

```
float4x4 WorldViewProj;

struct VS_OUTPUT
{
    float4 Pos : POSITION;
};

VS_OUTPUT VertexShader_Tutorial_1(float4 inPos : POSITION )
{
    VS_OUTPUT out;
    out.Pos = mul(inPos, WorldViewProj );
    return out;
};
```

See also

[Functions \(DirectX HLSL\)](#)

Signatures

Article • 08/23/2019 • 2 minutes to read

A shader signature is a list of the parameters that are either input to or output from a shader function. In Direct3D 10, adjacent stages effectively share a register array, where the output shader (or pipeline stage) writes data to specific locations in the register array and the input shader must read from the same locations. The API uses shader signatures to bind shader outputs with inputs without the overhead of semantic resolution.

In Direct3D 10, input signatures are generated from a shader-input declaration and the output signature is generated from a shader-output declaration. An input signature is said to be compatible with an output signature when the output signature is a strict subset (argument type and order match) of the input signature. The most straightforward way to achieve this is to link corresponding shader inputs and outputs by the same structure type.

Here is an example of compatible signatures.

```
// Vertex Shader Output Signature
Struct VSOut
{
    float4 Pos: SV_Position;
    float3 MyNormal: Normal;
    float2 MyTex : Texcoord0;
}

// Pixel Shader Input Signature
Struct PSInWorks
{
    float4 Pos: SV_Position;
    float3 MyNormal: Normal;
}
```

Here is an example of incompatible signatures; the order of parameters in the input signature does not match the order in the output signature.

```
// Vertex Shader Output Signature
Struct VSOut
{
    float4 Pos: SV_Position;
    float3 MyNormal: Normal;
```

```
    float2 MyTex : Texcoord0;  
}  
  
// Pixel Shader Input Signature  
Struct PSInFails  
{  
    float3 MyNormal: Normal;  
    float4 Pos: SV_Position;  
}
```

PSInWorks is a compatible subset of VSOut (the first two entries match both type and order with the first two entries in VSOut). However, PSInFails is incompatible because the ordering does not match with VSOut.

Related topics

[Functions](#)

Statements

Article • 08/23/2019 • 2 minutes to read

HLSL statements are combinations of variables and functions that make up expressions; like a sentence, an HLSL statement requires operators that determine how an expression will be evaluated. You may create individual statements in a shader, or create several statements in a group called a statement block. In order to create HLSL statements, you will need to know the syntax for writing expressions, and the operators that you will use to combine expressions.

- [Expressions](#)
- [Operators](#)
- [Statement Blocks](#)

Related topics

[Language Syntax \(DirectX HLSL\)](#)

Expressions

Article • 08/23/2019 • 2 minutes to read

An expression is a sequence of [variables](#) and literals, punctuated by [operators](#). A literal is an explicit data value, such as 1 for an integer or 2.1 for a floating-point number. Literals are often used to assign a value to a variable.

An expression followed by a semicolon (;) is called a statement. Statements range in complexity from simple expressions to blocks of statements that accomplish a sequence of actions. Flow-control statements determine the order statements are executed.

A statement block also indicates subscope. Variables declared within a statement block are only recognized within the block. HLSL statements determine the order in which expressions are evaluated. Each expression can be one of the following.

- An expression
- A [statement block](#)
- A [return statement](#)
- A [flow-control statement](#)

Related topics

[Statements \(DirectX HLSL\)](#)

Operators

Article • 06/30/2021 • 10 minutes to read

Expressions are sequences of [variables](#) and literals punctuated by [operators](#). Operators determine how the variables and literals are combined, compared, selected, and so on. The operators include:

Operator name	Operators
Additive and Multiplicative Operators	+, -, *, /, %
Array Operator	[i]
Assignment Operators	=, +=, -=, *=, /=, %=
Binary Casts	C rules for float and int, C rules or HLSL intrinsics for bool
Bitwise Operators	~, <<, >>, &, , ^, <<=, >>=, &=, =, ^=
Boolean Math Operators	& &, , ?:
Cast Operator	(type)
Comma Operator	,
Comparison Operators	<, >, ==, !=, <=, >=
Prefix or Postfix Operators	++, --
Structure Operator	.
Unary Operators	!, -, +

Many of the operators are per-component, which means that the operation is performed independently for each component of each variable. For example, a single component variable has one operation performed. On the other hand, a four-component variable has four operations performed, one for each component.

All of the operators that do something to the value, such as + and *, work per component.

The comparison operators require a single component to work unless you use the [all](#) or [any](#) intrinsic function with a multiple-component variable. The following operation fails because the if statement requires a single bool but receives a bool4:

```
if (A4 < B4)
```

The following operations succeed:

```
if ( any(A4 < B4) )
if ( all(A4 < B4) )
```

The binary cast operators [asfloat](#), [asint](#), and so on work per component except for [asddouble](#) whose special rules are documented.

Selection operators like period, comma, and array brackets do not work per component.

Cast operators change the number of components. The following cast operations show their equivalence:

```
(float) i4 -> float(i4.x)
(float4)i -> float4(i, i, i, i)
```

Additive and Multiplicative Operators

The additive and multiplicative operators are: +, -, *, /, %

```
int i1 = 1;
int i2 = 2;
int i3 = i1 + i2; // i3 = 3
i3 = i1 * i2; // i3 = 1 * 2 = 2
```

```
i3 = i1/i2; // i3 = 1/3 = 0.333. Truncated to 0 because i3 is an
integer.
i3 = i2/i1; // i3 = 2/1 = 2
```

```
float f1 = 1.0;
float f2 = 2.0f;
```

```
float f3 = f1 - f2; // f3 = 1.0 - 2.0 = -1.0
f3 = f1 * f2;      // f3 = 1.0 * 2.0 = 2.0
```

```
f3 = f1/f2;        // f3 = 1.0/2.0 = 0.5
f3 = f2/f1;        // f3 = 2.0/1.0 = 2.0
```

The modulus operator returns the remainder of a division. This produces different results when using integers and floating-point numbers. Integer remainders that are fractional will be truncated.

```
int i1 = 1;
int i2 = 2;
i3 = i1 % i2;      // i3 = remainder of 1/2, which is 1
i3 = i2 % i1;      // i3 = remainder of 2/1, which is 0
i3 = 5 % 2;         // i3 = remainder of 5/2, which is 1
i3 = 9 % 2;         // i3 = remainder of 9/2, which is 1
```

The modulus operator truncates a fractional remainder when using integers.

```
f3 = f1 % f2;      // f3 = remainder of 1.0/2.0, which is 0.5
f3 = f2 % f1;      // f3 = remainder of 2.0/1.0, which is 0.0
```

The % operator is defined only in cases where either both sides are positive or both sides are negative. Unlike C, it also operates on floating-point data types, as well as integers.

Array Operator

The array member selection operator "[i]" selects one or more components in an array. It is a set of square brackets that contain a zero-based index.

```
int arrayOfInts[4] = { 0,1,2,3 };
arrayOfInts[0] = 2;
arrayOfInts[1] = arrayOfInts[0];
```

The array operator can also be used to access a vector.

```
float4 4D_Vector = { 0.0f, 1.0f, 2.0f, 3.0f };  
float 1DFloat = 4D_Vector[1];           // 1.0f
```

By adding an additional index, the array operator can also access a matrix.

```
float4x4 mat4x4 = {{0,0,0,0}, {1,1,1,1}, {2,2,2,2}, {3,3,3,3} };  
mat4x4[0][1] = 1.1f;  
float 1DFloat = mat4x4[0][1];           // 0.0f
```

The first index is the zero-based row index. The second index is the zero-based column index.

Assignment Operators

The assignment operators are: `=`, `+=`, `-=`, `*=`, `/=`

Variables can be assigned literal values:

```
int i = 1;  
float f2 = 3.1f;  
bool b = false;  
string str = "string";
```

Variables can also be assigned the result of a mathematical operation:

```
int i1 = 1;  
i1 += 2;           // i1 = 1 + 2 = 3
```

A variable can be used on either side of the equals sign:

```
float f3 = 0.5f;  
f3 *= f3;           // f3 = 0.5 * 0.5 = 0.25
```

Division for floating-point variables is as expected because decimal remainders are not a problem.

```
float f1 = 1.0;  
f1 /= 3.0f;           // f1 = 1.0/3.0 = 0.333
```

Be careful if you are using integers that may get divided, especially when truncation affects the result. This example is identical to the previous example, except for the data type. The truncation causes a very different result.

```
int i1 = 1;  
i1 /= 3;           // i1 = 1/3 = 0.333, which gets truncated to 0
```

Binary Casts

Casting operation between int and float will convert the numeric value into the appropriate representations following C rules for truncating an int type. Casting a value from a float to an int and back to a float will result in a lossy conversion based on the precision of the target.

Binary casts may also be performed using [Intrinsic Functions \(DirectX HLSL\)](#), which reinterpret the bit representation of a number into the target data type.

```
asfloat() // Cast to float  
asint()   // Cast to int  
asuint()  // Cast to uint
```

Bitwise Operators

HLSL supports the following bitwise operators, which follow the same precedence as C with regard to other operators. The following table describes the operators.

 **Note**

Bitwise operators require [Shader Model 4_0](#) with Direct3D 10 and higher hardware.

Operator	Function
<code>~</code>	Logical Not
<code><<</code>	Left Shift
<code>>></code>	Right Shift
<code>&</code>	Logical And
<code> </code>	Logical Or
<code>^</code>	Logical Xor
<code><<=</code>	Left Shift Equal
<code>>>=</code>	Right Shift Equal
<code>&=</code>	And Equal
<code> =</code>	Or Equal
<code>^=</code>	Xor Equal

Bitwise operators are defined to operate only on int and uint data types. Attempting to use bitwise operators on float, or struct data types will result in an error.

Boolean Math Operators

The Boolean math operators are: `&&`, `||`, `?:`

```
bool b1 = true;
bool b2 = false;
bool b3 = b1 && b2 // b3 = true AND false = false
b3 = b1 || b2           // b3 = true OR false = true
```

Unlike short-circuit evaluation of `&&`, `||`, and `?:` in C, HLSL expressions never short-circuit an evaluation because they are vector operations. All sides of the expression are always evaluated.

Boolean operators function on a per-component basis. This means that if you compare two vectors, the result is a vector containing the Boolean result of the comparison for each component.

For expressions that use Boolean operators, the size and component type of each variable are promoted to be the same before the operation occurs. The promoted type determines the resolution at which the operation takes place, as well as the result type of the expression. For example an int3 + float expression would be promoted to float3 + float3 for evaluation, and its result would be of type float3.

Cast Operator

An expression preceded by a type name in parenthesis is an explicit type cast. A type cast converts the original expression to the data type of the cast. In general, the simple data types can be cast to the more complex data types (with a promotion cast), but only some complex data types can be cast into simple data types (with a demotion cast).

Only right hand side type casting is legal. For example, expressions such as `(int)myFloat = myInt;` are illegal. Use `myFloat = (float)myInt;` instead.

The compiler also performs implicit type cast. For example, the following two expressions are equivalent:

```
int2 b = int2(1,2) + 2;
int2 b = int2(1,2) + int2(2,2);
```

Comma Operator

The comma operator (,) separates one or more expressions that are to be evaluated in order. The value of the last expression in the sequence is used as the value of the sequence.

Here is one case worth calling attention to. If the constructor type is accidentally left off the right side of the equals sign, the right side now contains four expressions, separated by three commas.

```
// Instead of using a constructor
float4 x = float4(0,0,0,1);

// The type on the right side is accidentally left off
float4 x = (0,0,0,1);
```

The comma operator evaluates an expression from left to right. This reduces the right hand side to:

```
float4 x = 1;
```

HLSL uses scalar promotion in this case, so the result is as if this were written as follows:

```
float4 x = float4(1,1,1,1);
```

In this instance, leaving off the float4 type from the right side is probably a mistake that the compiler is unable to detect because this is a valid statement.

Comparison Operators

The comparison operators are: <, >, ==, !=, <=, >=.

Compare values that are greater than (or less than) any scalar value:

```
if( dot(lightDirection, normalVector) > 0 )
    // Do something; the face is lit
```

```
if( dot(lightDirection, normalVector) < 0 )
    // Do nothing; the face is backwards
```

Or, compare values equal to (or not equal to) any scalar value:

```
if(color.a == 0)
    // Skip processing because the face is invisible

if(color.a != 0)
    // Blend two colors together using the alpha value
```

Or combine both and compare values that are greater than or equal to (or less than or equal to) any scalar value:

```
if( position.z >= oldPosition.z )  
    // Skip the new face because it is behind the existing face
```

```
if( currentValue <= someInitialCondition )  
    // Reset the current value to its initial condition
```

Each of these comparisons can be done with any scalar data type.

To use comparison operators with vector and matrix types, use the [all](#) or [any](#) intrinsic function.

This operation fails because the if statement requires a single bool but receives a bool4:

```
if (A4 < B4)
```

These operations succeed:

```
if ( any(A4 < B4) )  
if ( all(A4 < B4) )
```

Prefix or Postfix Operators

The prefix and postfix operators are: `++`, `--`. Prefix operators change the contents of the variable before the expression is evaluated. Postfix operators change the contents of the variable after the expression is evaluated.

In this case, a loop uses the contents of `i` to keep track of the loop count.

```
float4 arrayOfFloats[4] = { 1.0f, 2.0f, 3.0f, 4.4f };  
  
for (int i = 0; i<4; )  
{  
    arrayOfFloats[i++] *= 2;  
}
```

Because the postfix increment operator (++) is used, `arrayOfFloats[i]` is multiplied by 2 before `i` is incremented. This could be slightly rearranged to use the prefix increment operator. This one is harder to read, although both examples are equivalent.

```
float4 arrayOfFloats[4] = { 1.0f, 2.0f, 3.0f, 4.4f };

for (int i = 0; i<4; )
{
    arrayOfFloats[++i - 1] *= 2;
}
```

Because the prefix operator (++) is used, `arrayOfFloats[i+1 - 1]` is multiplied by 2 after `i` is incremented.

The prefix decrement and postfix decrement operator (--) are applied in the same sequence as the increment operator. The difference is that decrement subtracts 1 instead of adding 1.

Structure Operator

The structure member selection operator (.) is a period. Given this structure:

```
struct position
{
    float4 x;
    float4 y;
    float4 z;
};
```

It can be read like this:

```
struct position pos = { 1,2,3 };

float 1D_Float = pos.x
1D_Float = pos.y
```

Each member can be read or written with the structure operator:

```
struct position pos = { 1,2,3 };
pos.x = 2.0f;
pos.z = 1.0f;           // z = 1.0f
pos.z = pos.x          // z = 2.0f
```

Unary Operators

The unary operators are: !, -, +

Unary operators operate on a single operand.

```
bool b = false;
bool b2 = !b;           // b2 = true
int i = 2;
int i2 = -i;           // i2 = -2
int j = +i2;           // j = +2
```

Operator Precedence

When an expression contains more than one operator, operator precedence determines the order of evaluation. Operator precedence for HLSL follows the same precedence as C.

Remarks

Curly braces ({},) start and end a statement block. When a statement block uses a single statement, the curly braces are optional.

Related topics

[Statements \(DirectX HLSL\)](#)

Statement Blocks

Article • 08/23/2019 • 2 minutes to read

A statement block contains one or more statements; each statement is made up of [expressions](#) and [operators](#).

```
{  
    statement 1;  
    statement 2;  
    ...  
    statement n;  
}
```

A statement block also indicates subscope. Variables declared within a statement block are only recognized within the block.

Related topics

[Statements \(DirectX HLSL\)](#)

Semantics

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

A semantic is a string attached to a shader input or output that conveys information about the intended use of a parameter. Semantics are required on all variables passed between shader stages. The syntax for adding a semantic to a shader variable is shown here ([Variable Syntax \(DirectX HLSL\)](#)).

In general, data passed between pipeline stages is completely generic and is not uniquely interpreted by the system; arbitrary semantics are allowed which have no special meaning. Parameters (in Direct3D 10 and later) which contain these special semantics are referred to as [System-Value Semantics](#).

Semantics Supported in Direct3D 9 and Direct3D 10 and later

The following types of semantics are supported in both Direct3D 9 and Direct3D 10 and later.

- [Vertex Shader Semantics](#)
- [Pixel Shader Semantics](#)

Vertex Shader Semantics

These semantics have meaning when attached to a vertex-shader parameter. These semantics are supported in both Direct3D 9 and Direct3D 10 and later.

Input	Description	Type
BINORMAL[n]	Binormal	float4
BLENDINDICES[n]	Blend indices	uint
BLENDWEIGHT[n]	Blend weights	float
COLOR[n]	Diffuse and specular color	float4
NORMAL[n]	Normal vector	float4
POSITION[n]	Vertex position in object space.	float4
POSITIONT	Transformed vertex position.	float4
PSIZE[n]	Point size	float
TANGENT[n]	Tangent	float4
TEXCOORD[n]	Texture coordinates	float4

Output	Description	Type
COLOR[n]	Diffuse or specular color	float4
FOG	Vertex fog	float
POSITION[n]	Position of a vertex in homogenous space. Compute position in screen-space by dividing (x,y,z) by w. Every vertex shader must write out a parameter with this semantic.	float4
PSIZE	Point size	float
TESSFACTOR[n]	Tessellation factor	float

n is an optional integer between 0 and the number of resources supported. For example, POSITION0, TEXCOORD1, etc.

Pixel Shader Semantics

These semantics have meaning when attached to a pixel-shader input parameter. These semantics are supported in both Direct3D 9 and Direct3D 10 and later.

Input	Description	Type
COLOR[n]	Diffuse or specular color.	float4
TEXCOORD[n]	Texture coordinates	float4
VFACE	Floating-point scalar that indicates a back-facing primitive. A negative value faces backwards, while a positive value faces the camera.	float
	<p>[!Note]</p> <p>This semantic is available in Direct3D 9 Shader Model 3.0. For Direct3D 10 and later, use SV_IsFrontFace instead.</p>	
VPOS	The pixel location (x,y) in screen space. To convert a Direct3D 9 shader (that uses this semantic) to a Direct3D 10 and later shader, see Direct3D 9 VPOS and Direct3D 10 SV_Position	float2

Output	Description	Type
COLOR[n]	Output color	float4
DEPTH[n]	Output depth	float

n is an optional integer between 0 and the number of resources supported. For example, PSIZE0, COLOR1, etc.

The COLOR semantic is only valid in shader compatibility mode (that is, when the shader is created using D3D10_SHADER_ENABLE_BACKWARDS_COMPATIBILITY).

Semantics Supported Only for Direct3D 10 and Newer.

The following types of semantics have been newly introduced for Direct3D 10 and are not available to Direct3D 9.

- [System-Value Semantics](#)

System-Value Semantics

System-value semantics are new to Direct3D 10. All system-values begin with an SV_ prefix, a common example is SV_POSITION, which is interpreted by the rasterizer stage. The system-values are valid at other parts of the pipeline. For instance, SV_Position can be specified as an input to a vertex shader as well as an output. Pixel shaders can only write to parameters with the SV_Depth and SV_Target system-value semantics.

Other system values (SV_VertexID, SV_InstanceID, SV_IsFrontFace) can only be input into the first active shader in the pipeline that can interpret the particular value; after that the shader function must pass the values to subsequent stages.

SV_PrimitiveID is an exception to this rule of only being input into the first active shader in the pipeline that can interpret the particular value; the hardware can provide the same ID value as input to the hull-shader stage, domain-shader stage, and after that whichever stage is the first enabled: geometry-shader stage or pixel-shader stage.

If tessellation is enabled, the hull-shader stage and domain-shader stage are present. For a given patch, the same PrimitiveID applies to the patch's hull-shader invocation, and all tessellated domain shader invocations. The same PrimitiveID also propagates to the next active stage: geometry-shader stage or pixel-shader stage if enabled.

If the geometry shader inputs SV_PrimitiveID and because it can output zero or one or more primitives per invocation, the shader must program its own choice of SV_PrimitiveID value for each output primitive if a subsequent pixel shader inputs SV_PrimitiveID.

As another example, SV_PrimitiveID cannot be interpreted by the vertex-shader stage because a vertex can be a member of multiple primitives.

These semantics have been added to Direct3D 10; they are not available in Direct3D 9.

System-value semantics for the rasterizer stage.

System-Value Semantic	Description	Type
SV_ClipDistance[n]	Clip distance data. SV_ClipDistance values are each assumed to be a float32 signed distance to a plane. Primitive setup only invokes rasterization on pixels for which the interpolated plane distance(s) are ≥ 0 . Multiple clip planes can be implemented simultaneously, by declaring multiple component(s) of one or more vertex elements as the SV_ClipDistance. The combined clip and cull distance values are at most D3D#_CLIP_OR_CULL_DISTANCE_COUNT components in at most D3D#_CLIP_OR_CULL_DISTANCE_ELEMENT_COUNT registers. Available to all shaders to be read or written to, except the vertex shader which can write the value but not take it as input. The clipplanes attribute works like SV_ClipDistance but works on all hardware feature level 9_x and higher. For more info, see User clip planes on feature level 9 hardware .	float
SV_CullDistance[n]	Cull distance data. When component(s) of vertex Element(s) are given this label, these values are each assumed to be a float32 signed distance to a plane. Primitives will be completely discarded if the plane distance(s) for all of the vertices in the primitive are < 0 . Multiple cull planes can be used simultaneously, by declaring multiple component(s) of one or more vertex elements as the SV_CullDistance. The combined clip and cull distance values are at most D3D#_CLIP_OR_CULL_DISTANCE_COUNT components in at most D3D#_CLIP_OR_CULL_DISTANCE_ELEMENT_COUNT registers. Available to all shaders to be read or written to, except the vertex shader which can write the value but not take it as input.	float
SV_Coverage	A mask that can be specified on input, output, or both of a pixel shader. For SV_Coverage on a pixel shader, OUTPUT is supported on ps_4_1 or higher. For SV_Coverage on a pixel shader, INPUT requires ps_5_0 or higher.	uint
SV_Depth	Depth buffer data. Can be written by pixel shader.	float
SV_DepthGreaterEqual	In a pixel shader, allows outputting depth, as long as it is greater than or equal to the value determined by the rasterizer. Enables adjusting depth without disabling early Z.	float
SV_DepthLessEqual	In a pixel shader, allows outputting depth, as long as it is less than or equal to the value determined by the rasterizer. Enables adjusting depth without disabling early Z.	float
SV_DispatchThreadID	Defines the global thread offset within the Dispatch call, per dimension of the group. Available as input to compute shader. (read only)	uint3
SV_DomainLocation	Defines the location on the hull of the current domain point being evaluated. Available as input to the domain shader. (read only)	float2 3
SV_GroupID	Defines the group offset within a Dispatch call, per dimension of the dispatch call. Available as input to the compute shader. (read only)	uint3
SV_GroupIndex	Provides a flattened index for a given thread within a given group. Available as input to the compute shader. (read only)	uint
SV_GroupThreadID	Defines the thread offset within the group, per dimension of the group. Available as input to the compute shader. (read only)	uint3
SV_GSInstanceID	Defines the instance of the geometry shader. Available as input to the geometry shader. The instance is needed as a geometry shader can be invoked up to 32 times on the same geometry primitive.	uint
SV_InnerCoverage	Represents underestimated conservative rasterization information (i.e. whether a pixel is guaranteed-to-be-fully covered). Can be read or written by the pixel shader.	

System-Value Semantic	Description	Type
SV_InsideTessFactor	Defines the tessellation amount within a patch surface. Available in the hull shader for writing, and available in the domain shader for reading.	float[float[2]]
SV_InstanceID	Per-instance identifier automatically generated by the runtime (see Using System-Generated Values (Direct3D 10)). Available to all shaders.	
SV_IsFrontFace	Specifies whether a triangle is front facing. For lines and points, IsFrontFace has the value true. The exception is lines drawn out of triangles (wireframe mode), which sets IsFrontFace the same way as rasterizing the triangle in solid mode. Can be written to by the geometry shader, and read by the pixel shader.	bool
SV_OutputControlPointID	Defines the index of the control point ID being operated on by an invocation of the main entry point of the hull shader. Can be read by the hull shader only.	uint
SV_Position	When SV_Position is declared for input to a shader, it can have one of two interpolation modes specified: linearNoPerspective or linearNoPerspectiveCentroid, where the latter causes centroid-snapped xyzw values to be provided when multisample antialiasing. When used in a shader, SV_Position describes the pixel location. Available in all shaders to get the pixel center with a 0.5 offset.	float4
SV_PrimitiveID	Per-primitive identifier automatically generated by the runtime (see Using System-Generated Values (Direct3D 10)). Can be written to by the geometry or pixel shaders, and read by the geometry, pixel, hull or domain shaders.	uint
SV_RenderTargetArrayIndex	Render-target array index. Applied to geometry shader output, and indicates the render target array slice that the primitive will be drawn to by the pixel shader. SV_RenderTargetArrayIndex is valid only if the render target is an array resource. This semantic applies only to primitives; if a primitive has more than one vertex, then the value from the leading vertex is used. This value also indicates which array slice of a depth/stencil view is used for read/write purposes. Can be written from the geometry shader, and read by the pixel shader. If D3D11_FEATURE_DATA_D3D11_OPTIONS3::VPAndRTArrayIndexFromAnyShaderFeedingRasterizer is <code>true</code> , then SV_RenderTargetArrayIndex is applied to any shader feeding the rasterizer.	uint
SV_SampleIndex	Sample frequency index data. Available to be read or written to by the pixel shader only.	uint
SV_StencilRef	Represents the current pixel shader stencil reference value. Can be written by the pixel shader only.	uint
SV_Target[n], where 0 <= n <= 7	The output value that will be stored in a render target. The index indicates which of the 8 possibly bound render targets to write to. The value is available to all shaders.	float[2 3 4]
SV_TessFactor	Defines the tessellation amount on each edge of a patch. Available for writing in the hull shader and reading in the domain shader.	float[2 3 4]
SV_VertexID	Per-vertex identifier automatically generated by the runtime (see Using System-Generated Values (Direct3D 10)). Available as the input to the vertex shader only.	uint
SV_ViewportArrayIndex	Viewport array index. Applied to geometry shader output, and indicates which viewport to use for the primitive currently being written out. Can be read by the pixel shader. The primitive will be transformed and clipped against the viewport specified by the index before it is passed to the rasterizer. This semantic applies only to primitives; if a primitive has more than one vertex, then the value from the leading vertex is used. If D3D11_FEATURE_DATA_D3D11_OPTIONS3::VPAndRTArrayIndexFromAnyShaderFeedingRasterizer is <code>true</code> , then SV_ViewportArrayIndex is applied to any shader feeding the rasterizer.	uint
SV_ShadingRate	Defines, through shading rate values , the number of pixels written by one pixel shader invocation for Variable Shading Rate Tier 2 or higher devices. Can be read from the pixel shader. Can be written from a vertex or geometry shader.	uint

Limitations when writing SV_Depth:

- When multisampling (MultisampleEnable is TRUE in [D3D10_RASTERIZER_DESC](#)) and writing a depth value (using a pixel shader), the single value written out is also used in the [depth test](#); so the ability to render primitive edges at higher resolution is lost when multisampling.
- When using dynamic-flow control, it is impossible to determine at compile time whether a shader that writes SV_Depth in some paths will be guaranteed to write SV_Depth in every execution. Failure to write SV_Depth when declared results in undefined behavior (which may or may not include discard of the pixel).
- Any float32 value including +/-INF and NaN can be written to SV_Depth.
- Writing SV_Depth is still valid when performing Dual Source Color Blending.

Migration from Direct3D 9 to Direct3D 10 and later

The following issues should be considered when migrating code from Direct3D 9 to Direct3D 10 and later:

Mapping to Direct3D 9 Semantics

A few of the Direct3D 10 and later semantics map directly to Direct3D 9 semantics.

Direct3D 10 Semantic	Direct3D 9 Equivalent Semantic
SV_Depth	DEPTH
SV_Position	POSITION
SV_Target	COLOR

[!] Note to Direct3D 9 developers: For Direct3D 9 targets, shader semantics must map to valid Direct3D 9 semantics. For backwards compatibility POSITION0 (and its variant names) is treated as SV_Position, COLOR is treated as SV_TARGET.

- [Mapping to Direct3D 9 Semantics](#)
- [Direct3D 9 VPOS and Direct3D 10 SV_Position](#)
- [User clip planes in HLSL](#)

Direct3D 9 VPOS and Direct3D 10 SV_Position

The D3D10 semantic SV_Position provides similar functionality to the Direct3D 9 shader model 3 VPOS semantic. For instance, in Direct3D 9 the following syntax is used for a pixel shader using screen space coordinates:

```
HLSL

float4 psMainD3D9( float4 screenSpace : VPOS ) : COLOR
{
    // code here
}
```

VPOS was added for shader model 3 support, to specify screen space coordinates, since the POSITION semantic was intended for object-space coordinates.

In Direct3D 10 and later, the SV_Position semantic (when used in the context of a pixel shader) specifies screen space coordinates (offset by 0.5). Therefore, the Direct3D 9 shader would be roughly equivalent (without accounting for the 0.5 offset) to the following:

```
HLSL

float4 psMainD3D10( float4 screenSpace : SV_Position ) : COLOR
{
```

```
// code here  
}
```

When migrating from Direct3D 9 to Direct3D 10 and later, you will need to be aware of this when translating your shaders.

User clip planes in HLSL

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](#).

Related topics

- [Language Syntax](#)
- [Variables \(DirectX HLSL\)](#)

SV_DispatchThreadID

Article • 06/08/2021 • 2 minutes to read

Indices for which combined thread and thread group a compute shader is executing in. SV_DispatchThreadID is the sum of SV_GroupID * numthreads and GroupThreadID. It varies across the range specified in [Dispatch](#) and [numthreads](#). For example if Dispatch(2,2,2) is called on a compute shader with numthreads(3,3,3) SV_DispatchThreadID will have a range of 0..5 for each dimension.

Type

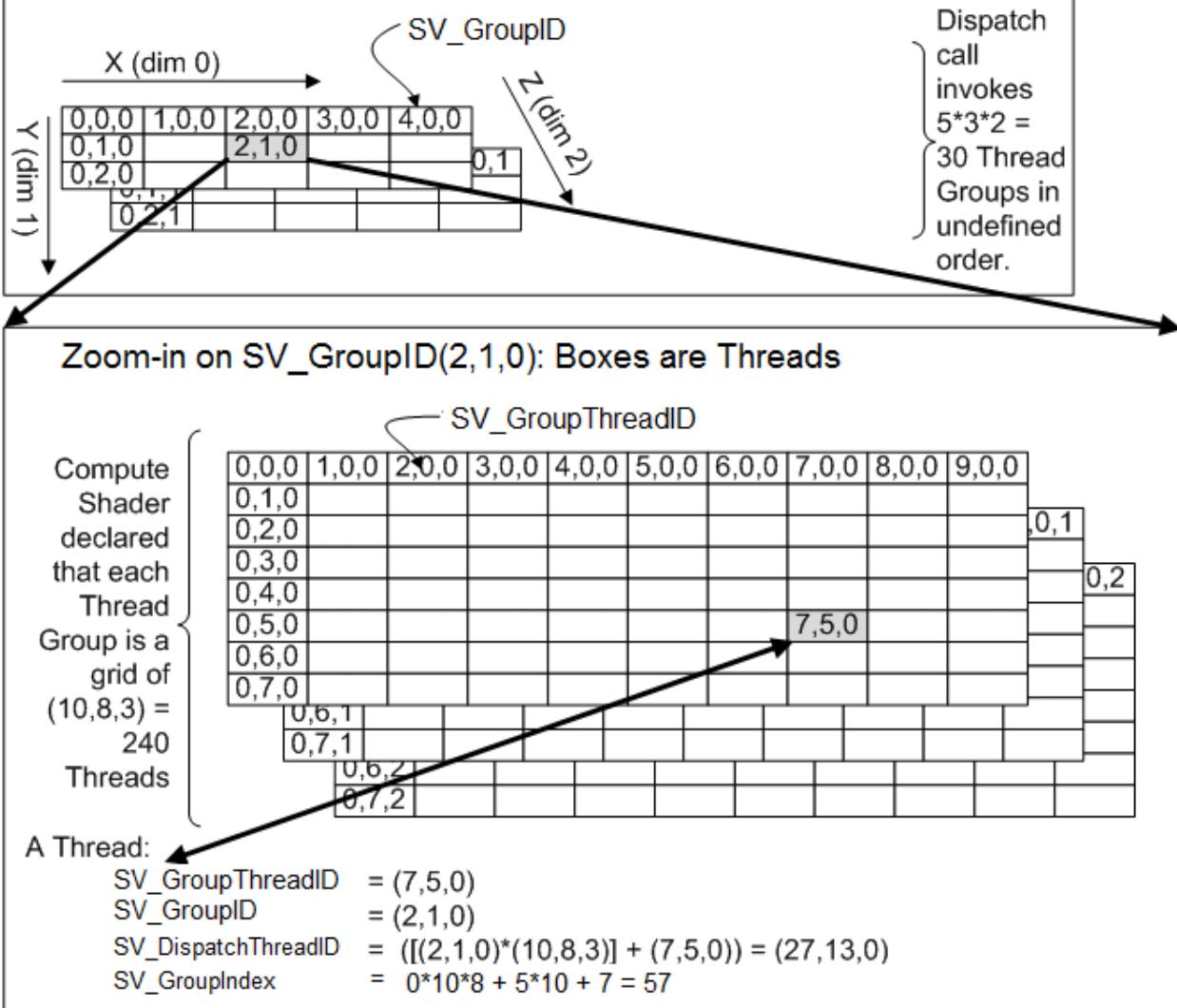
Type
uint3

Remarks

This system value is optional.

The following illustration shows the relationship between the parameters passed to [Dispatch](#), Dispatch(5,3,2), the values specified in the [numthreads](#) attribute, numthreads(10,8,3), and values that will be passed to the compute shader for the thread-related system values ([SV_GroupIndex](#), [SV_DispatchThreadID](#), [SV_GroupThreadID](#), [SV_GroupID](#)).

Dispatch(5,3,2) : Each box below is a Thread Group



This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	

See also

[Semantics](#)

[Shader Model 5](#)

SV_DomainLocation

Article • 06/08/2021 • 2 minutes to read

Defines the location on the hull of the current domain point being evaluated.

Type

Type	Input topology
float2	quad patch
float3	tri patch
float2	isoline

Remarks

This system value is required.

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
		x			

See also

[Semantics](#)

[Shader Model 5](#)

SV_TessFactor

Article • 06/30/2021 • 2 minutes to read

Defines the tessellation amount on each edge of a patch.

Type

Type	Input topology
float[4]	quad patch
float[3]	tri patch
float[2]	isoline

Tessellation factors must be declared as an array; they cannot be packed into a single vector.

Remarks

The value for tessellation factor must be defined during the patch constant function of the hull shader.

Required output value for the hull shader if using quad or tri patches. This value is also a required input value for the domain shader to match the patch-constant data signatures between the tessellation stages.

For an isoline, the first value in SV_TessFactor is the line-density tessellation factor, the second value is the line-detail tessellation factor.

Tri Patch Tessellation Factors

The first component provides the tessellation factor for the $u==0$ edge of the patch. The second component provides the tessellation factor for the $v==0$ edge of the patch. The third component provides the tessellation factor for the $w==0$ edge of the patch.

Quad Patch Tessellation Factors

The first component provides the tessellation factor for the $u==0$ edge of the patch. The second component provides the tessellation factor for the $v==0$ edge of the patch. The third component provides the tessellation factor for the $u==1$ edge of the patch. The fourth component provides the tessellation factor for the $v==1$ edge of the patch. The ordering of the edges is clockwise, starting from the $u==0$ edge, which is the left side of the patch, and from the $v==0$ edge, which is the top of the patch.

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
	x	x			

See also

[Semantics](#)

[Shader Model 5](#)

SV_GroupID

Article • 06/08/2021 • 2 minutes to read

Indices for which thread group a compute shader is executing in. The indices are to the whole group and not an individual thread. Possible values vary across the range passed as parameters to [Dispatch](#). For example calling Dispatch(2,1,1) results in possible values of 0,0,0 and 1,0,0.

Defines the group offset within a [Dispatch](#) call, per dimension of the dispatch call.

Type

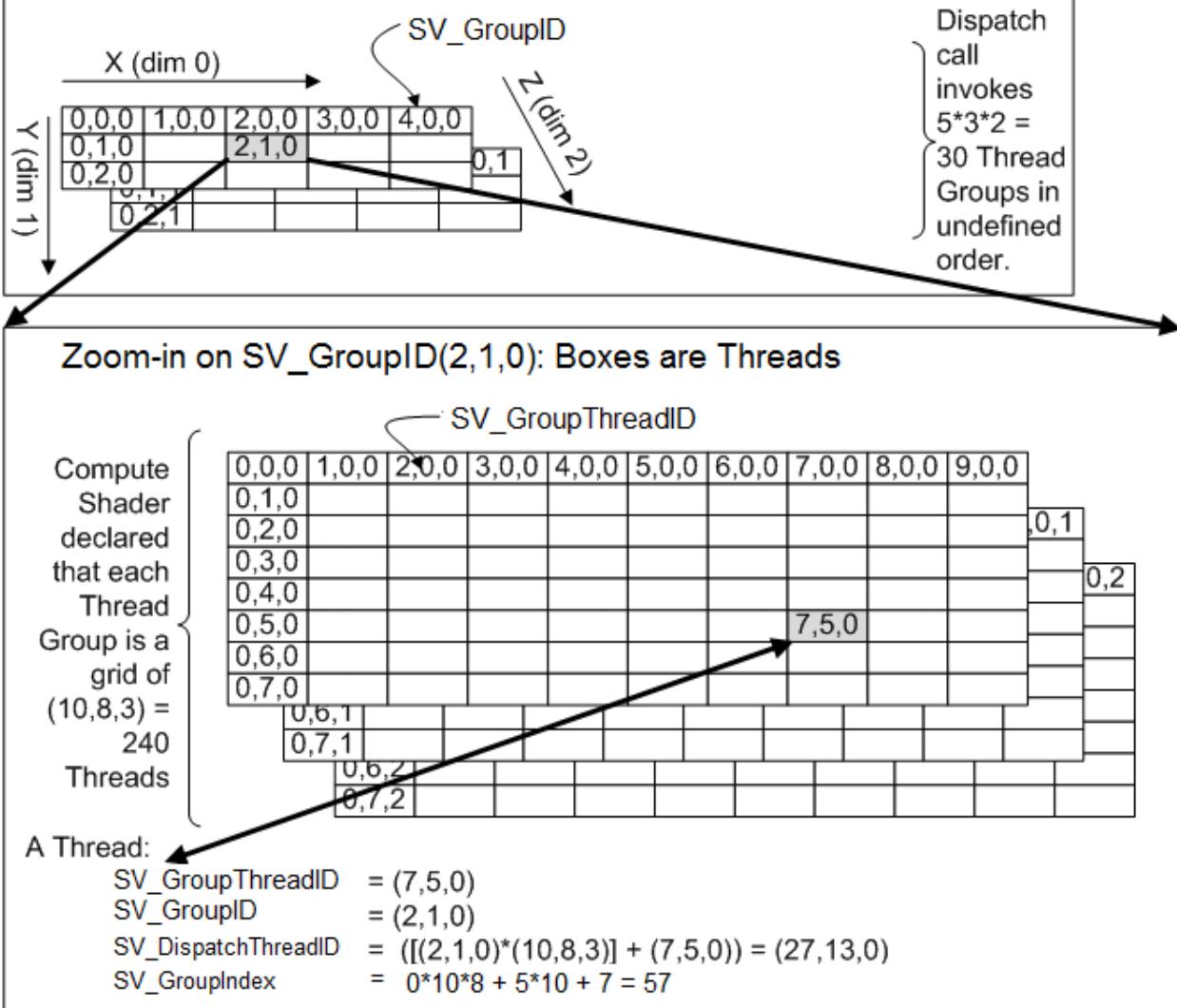
Type
uint3

Remarks

This system value is optional.

The following illustration shows the relationship between the parameters passed to [Dispatch](#), Dispatch(5,3,2), the values specified in the [numthreads](#) attribute, numthreads(10,8,3), and values that will be passed to the compute shader for the thread-related system values ([SV_GroupIndex](#),[SV_DispatchThreadID](#),[SV_GroupThreadID](#),[SV_GroupID](#)).

Dispatch(5,3,2) : Each box below is a Thread Group



This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	

See also

[Semantics](#)

[Shader Model 5](#)

SV_GroupIndex

Article • 08/19/2020 • 2 minutes to read

The "flattened" index of a compute shader thread within a thread group, which turns the multi-dimensional SV_GroupThreadID into a 1D value. SV_GroupIndex varies from 0 to $(\text{numthreadsX} * \text{numthreadsY} * \text{numThreadsZ}) - 1$.

Type

Type
uint

Remarks

```
SV_GroupIndex = SV_GroupThreadID.z*dimx*dimy +
                SV_GroupThreadID.y*dimx +
                SV_GroupThreadID.x
```

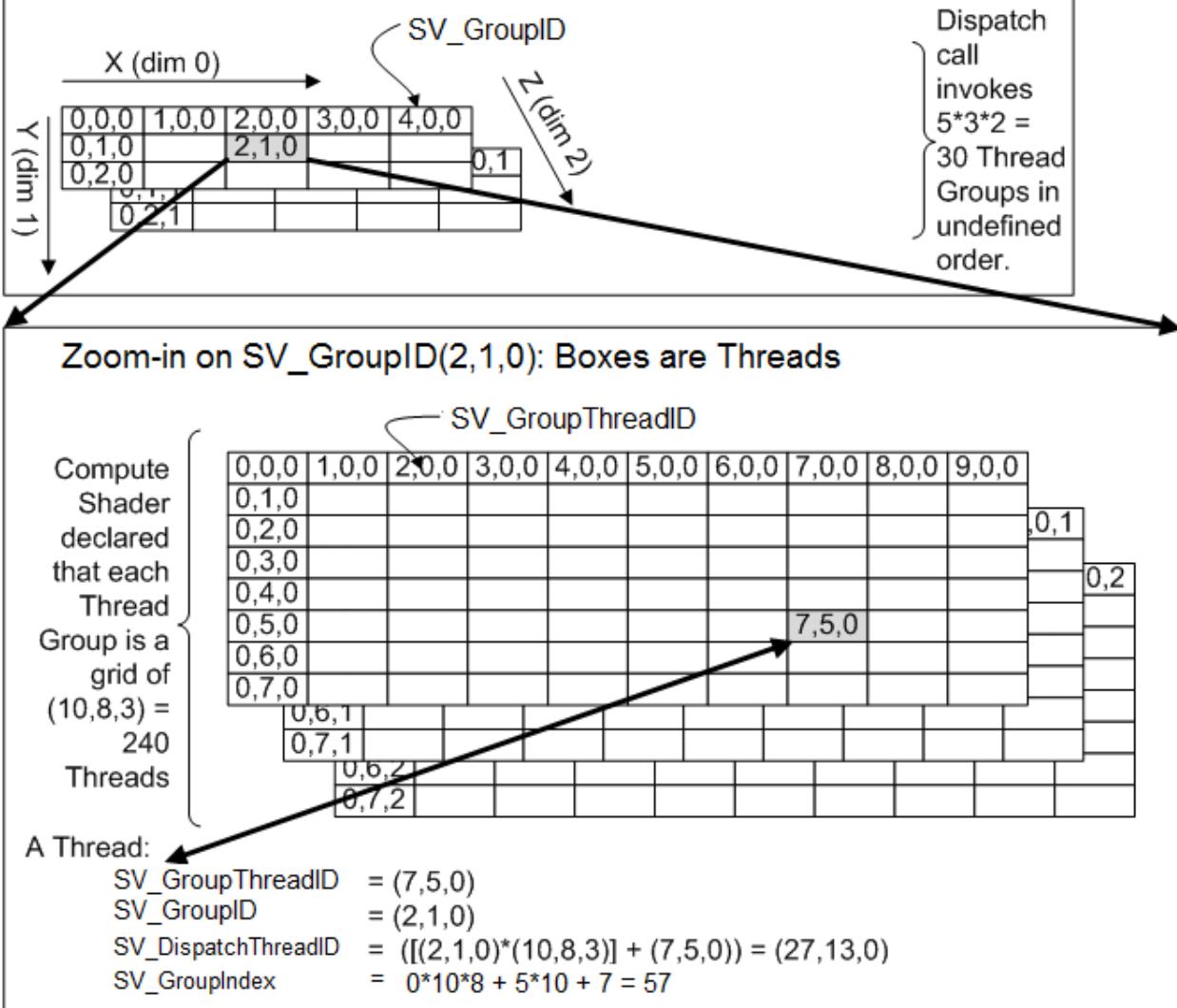
where dimx and dimy are the dimensions specified in the [numthreads](#) attribute for the entry point.

This system value is optional. However, its use ensures that a thread only writes to its assigned region of memory in the groupshared variable.

The following illustration shows the relationship between the parameters passed to [ID3D11DeviceContext::Dispatch](#), Dispatch(5,3,2), the values specified in the [numthreads](#) attribute, numthreads(10,8,3), and values that will be passed to the compute shader for the thread-related system values

(SV_GroupIndex, [SV_DispatchThreadID](#), [SV_GroupThreadID](#), [SV_GroupID](#)).

Dispatch(5,3,2) : Each box below is a Thread Group



This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	

See also

[Semantics](#)

[Shader Model 5](#)

SV_GroupThreadID

Article • 06/08/2021 • 2 minutes to read

Indices for which an individual thread within a thread group a compute shader is executing in. SV_GroupThreadID varies across the range specified for the compute shader in the [numthreads](#) attribute. For example if numthreads(3,2,1) was specified possible values for the SV_GroupThreadID input value have this range of values (0-2,0-1,0).

Type

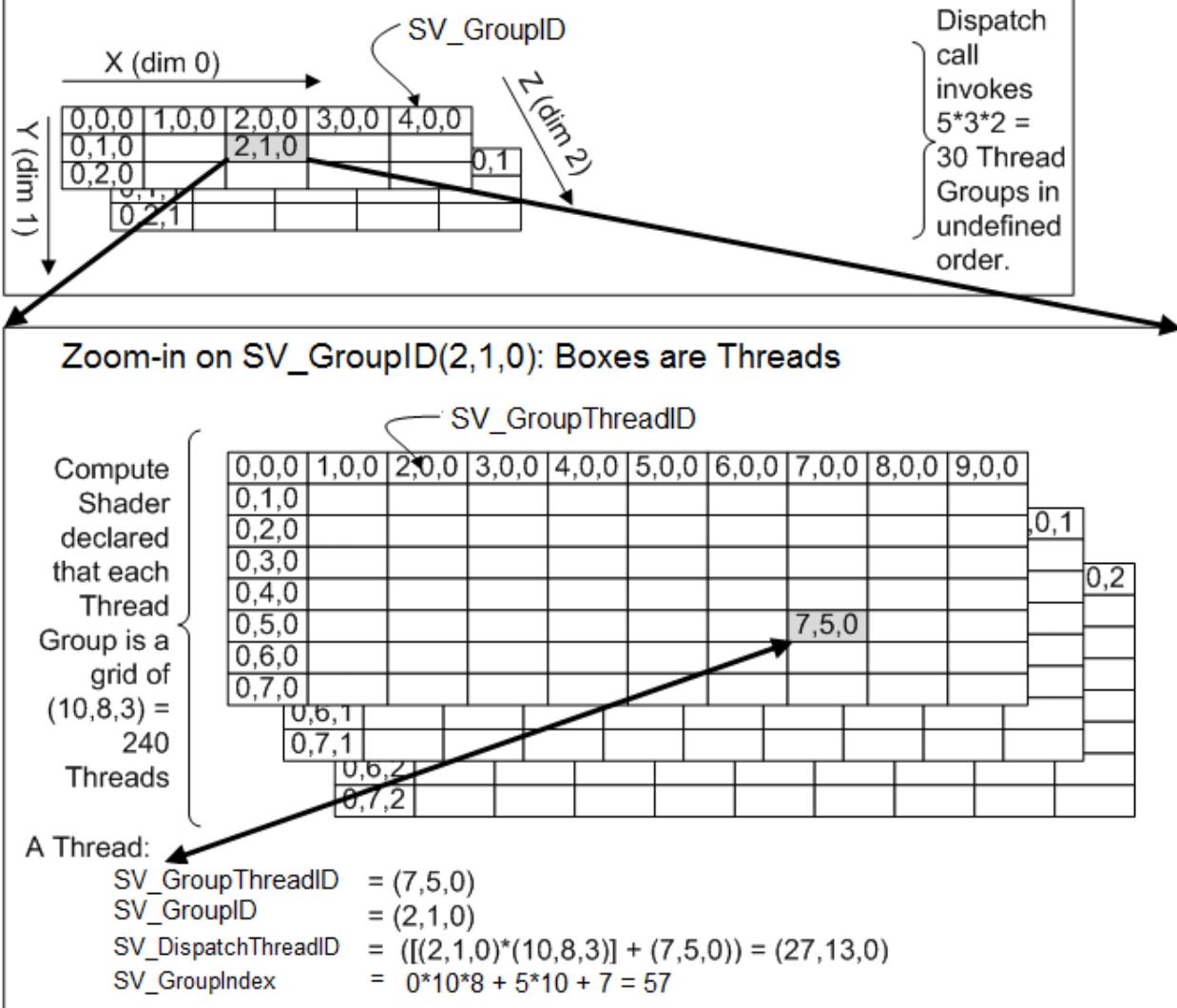
Type
uint3

Remarks

This system value is optional, and is always within the bounds of the values passed into the [numthreads](#) attribute.

The following illustration shows the relationship between the parameters passed to [Dispatch](#), Dispatch(5,3,2), the values specified in the [numthreads](#) attribute, numthreads(10,8,3), and values that will be passed to the compute shader for the thread-related system values ([SV_GroupIndex](#),[SV_DispatchThreadID](#),[SV_GroupThreadID](#),[SV_GroupID](#)).

Dispatch(5,3,2) : Each box below is a Thread Group



This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	

See also

[Semantics](#)

[Shader Model 5](#)

SV_GSInstanceID

Article • 06/08/2021 • 2 minutes to read

Defines the [instance](#) of the geometry shader.

Type

Type
uint

Remarks

This system value is optional.

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
			x		

See also

[Semantics](#)

[Shader Model 5](#)

SV_InsideTessFactor

Article • 06/08/2021 • 2 minutes to read

Defines the tessellation amount within a patch surface.

Type

Type	Input topology
float[2]	quad patch
float	tri patch
unused	isoline

Tessellation factors must be declared as array; they cannot be packed into a single vector.

Remarks

This value must be defined during the patch constant function of the hull shader.

Required output value for the hull shader if using quad or tri patches. This value is a required input for the domain shader in order for hardware to match the signatures through the tessellator.

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
	x	x			

See also

[Semantics](#)

[Shader Model 5](#)

SV_OutputControlPointID

Article • 06/08/2021 • 2 minutes to read

Defines the index of the control point ID being operated on by an invocation of the main entry point of the hull shader.

Type

Type
uint

Remarks

This system value is optional.

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
	x				

See also

[Semantics](#)

[Shader Model 5](#)

Shader Models vs Shader Profiles

Article • 06/30/2021 • 2 minutes to read

The High Level Shading Language for DirectX implements a series of shader models. Using HLSL, you can create C-like programmable shaders for the Direct3D pipeline. Each shader model builds on the capabilities of the model before it, implementing more functionality with fewer restrictions.

Shader model 1 started with DirectX 8 and included assembly level and C-like instructions. This model has many limitations caused by early programmable shader hardware. Shader model 2 and 3 greatly expanded on the number of instructions, and constants shaders could use. They are much more powerful than shader model 1, but still carry some of the existing limitations of the first shader model.

Starting with Windows Vista, shader model 4 is a complete redesign. It allows unlimited instructions and constants (within hardware constraints of your machine), has templated objects to make texture sampling cleaner and more efficient, and has the fewest restrictions of any shader model. It does however require the Windows Driver Model which is only available on the Windows Vista (or later) operating system.

Shader Profiles

A shader profile is the target for compiling a shader; this table lists the shader profiles that are supported by each shader model.

Shader model	Shader profiles
Shader Model 1	vs_1_1
Shader Model 2	ps_2_0, ps_2_x, vs_2_0, vs_2_x, ps_4_0_level_9_0, ps_4_0_level_9_1, ps_4_0_level_9_3, vs_4_0_level_9_0, vs_4_0_level_9_1, vs_4_0_level_9_3, lib_4_0_level_9_1, lib_4_0_level_9_3
Shader Model 3	ps_3_0, vs_3_0
Shader Model 4	cs_4_0, gs_4_0, ps_4_0, vs_4_0, cs_4_1, gs_4_1, ps_4_1, vs_4_1, lib_4_0, lib_4_1

Shader	Shader profiles
model	
Shader	cs_5_0, ds_5_0, gs_5_0, hs_5_0, ps_5_0, vs_5_0, lib_5_0 (Although gs_4_0, gs_4_1, ps_4_0, ps_4_1, vs_4_0, and vs_4_1 were introduced in shader model 4.0, shader model 5 adds support to these shader profiles for structured buffers and byte address buffers.)
Model	
6	cs_6_0, ds_6_0, gs_6_0, hs_6_0, ps_6_0, vs_6_0, lib_6_0

Differences between Direct3D 9 and Direct3D 10:

- Direct3D 9 introduced shader models 1, 2, and 3.
- Direct3D 10 introduced shader model 4.
- Direct3D 10.1 introduced shader model 4.1.

Effect Profiles

An effect profile is the target for compiling an effect/shader; this table lists the effect profiles that are supported by each version of Direct3D.

Differences between Direct3D 9 and Direct3D 10:

- Direct3D 9 introduced effect-framework profiles fx_1_0 and fx_2_0.
- Direct3D 10 introduced effect-framework profile fx_4_0.
- Direct3D 10.1 introduced effect-framework profile fx_4_1.
- Direct3D 11 introduced effect-framework profile fx_5_0.

Note

These legacy effects profiles are deprecated.

Related topics

[Reference for HLSL](#)

Shader Model 1

Article • 03/16/2022 • 2 minutes to read

Shader Model 1 was the first shader model created in DirectX. It introduced vertex and pixel shaders to the first implementation of the programmable pipeline.

Feature	Capability
Instruction Set	<ul style="list-style-type: none">• HLSL functions• Vertex shader assembly instructions (see Instructions - vs_1_1). Support for pixel shader instructions (ps_1_x) has been deprecated. To compile ps_1_x shaders as ps_2_0 shaders see Compiling Shader Model 1.
Register Set	<ul style="list-style-type: none">• Vertex shader registers
Vertex Shader Max	128 instructions
Shader Profiles	vs_1_1

For more details on shader model 1, see:

- [Vertex Shader](#)

Related topics

[Shader Models vs Shader Profiles](#)

Shader Model 2

Article • 03/16/2022 • 2 minutes to read

Shader Model 2 added additional capabilities to [shader model 1](#).

Feature	Capability
Instruction Set	<ul style="list-style-type: none">• HLSL functions• Assembly instructions (see Instructions - vs_2_0, Instructions - vs_2_x, ps_2_0 Instructions, ps_2_x Instructions)
Register Set	<ul style="list-style-type: none">• Pixel shader registers (see ps_2_0 Registers, ps_2_x Registers)• Vertex shader registers (see Registers - vs_2_0, Registers - vs_2_x)
Pixel Shader Max	<ul style="list-style-type: none">• ps_2_0 - 32 texture + 64 arithmetic• ps_2_x - 96 minimum, and up to the number of slots in D3DCAPS9.D3DPSHADERCAPS2_0.NumInstructionSlots. See D3DPSHADERCAPS2_0
Vertex Shader Max	256 instructions
Shader Profiles	ps_2_0, ps_2_x, vs_2_0, vs_2_x

For more details on shader model 2, see:

- [Pixel Shader 2.0](#), [Pixel Shader 2.x](#)
- [Vertex Shader 2.0](#), [Vertex Shader 2.x](#)

Related topics

[Shader Models vs Shader Profiles](#)

Shader Model 3 (HLSL reference)

Article • 03/16/2022 • 2 minutes to read

Shader Model 3 added additional capabilities to [shader model 2](#).

Feature	Capability
Instruction Set	<ul style="list-style-type: none">• HLSL functions• Assembly instructions (see ps_3_0 Instructions, Instructions - vs_3_0)
Register Set	<ul style="list-style-type: none">• Pixel shader registers (see ps_3_0 Registers)• Vertex shader registers (see Registers - vs_3_0)
Pixel Shader Max	512 minimum, and up to the number of slots in D3DCAPS9.MaxPixelShader30InstructionSlots (see D3DPSHADERCAPS2_0).
Vertex Shader Max	512 minimum, and up to the number of slots in D3DCAPS9.MaxVertexShader30InstructionSlots (see D3DCAPS9).
Shader Profiles	ps_3_0, vs_3_0

For more details on model 3 shaders, see:

- [Pixel Shader 3.0](#)
- [Vertex Shader 3.0](#)

Related topics

[Shader Models vs Shader Profiles](#)

Shader model 3 (HLSL reference)

Article • 06/08/2021 • 9 minutes to read

Vertex shaders and pixel shaders are simplified considerably from earlier shader versions. If you are implementing shaders in hardware, you may not use `vs_3_0` or `ps_3_0` with any other shader versions, and you may not use either shader type with the fixed function pipeline. These changes make it possible to simplify drivers and the runtime. The only exception is that software-only `vs_3_0` shaders may be used with any pixel shader version. In addition, if you are using a software-only `vs_3_0` shader with a previous pixel shader version, the vertex shader can only use output semantics that are compatible with flexible vertex format (FVF) codes.

The semantics used on vertex shader outputs must be used on pixel shader inputs. The semantics are used to map the vertex shader outputs to the pixel shader inputs, similar to the way the vertex declaration is mapped to the vertex shader input registers and previous shader models. See [Match Semantics on vs 3.0 and ps 3.0 Shaders](#).

Additional wrap mode render states have been added to cover the possibility of additional texture coordinates in this new scheme. Attributes with `D3DDECLUSAGE_TEXCOORD` and usage index from 0 to 15 are interpolated in wrap mode when the corresponding [`D3DRS_WRAP*`](#) is set.

- [Vertex Shader Model 3 Features](#)
- [Pixel Shader Model 3 Features](#)
- [Match Semantics on vs_3_0 and ps_3_0 Shaders](#)
- [Fog, Depth, and Shading Mode Changes](#)
- [Floating Point and Integer Conversions](#)
- [Specifying Full or Partial Precision](#)
- [Software Vertex and Pixel Shaders](#)

Vertex Shader Model 3 Features

The vertex shader output register types have been collapsed into twelve registers (see [Output Registers](#)). Each register that is used needs to be declared using the `dcl` instruction and a semantic (for example, `dcl_color0 o0.xyzw`).

The `3_0` vertex shader model (`vs_3_0`) expands on the features of `vs_2_0` with more powerful register indexing, a set of simplified output registers, the ability to sample a texture in a vertex shader, and the ability to control the rate at which shader inputs are initialized.

Index Any Register

All registers ([Input Register](#) and [Output Registers](#)) can be indexed using [Loop Counter Register](#) (only constant registers could be indexed in earlier versions.)

You must declare input and output registers before indexing them. However, you may not index any output register that has been declared with a position or point size semantic. In fact, if indexing is used the position and psize semantics have to be declared in the o0 and o1 registers respectively.

You are only allowed to index a continuous range of registers; that is, you cannot index across registers that have not been declared. While this restriction may be inconvenient, it permits hardware optimization to take place. Attempting to index across non-contiguous registers will produce undefined results. Shader validation does not enforce this restriction.

Simplify Output Registers

All the various types of output registers have been collapsed into twelve output registers: 1 for position, 2 for color, 8 for texture, and 1 for fog or point size. These registers will interpolate any data they contain for the pixel shader. Output register declarations are required and semantics are assigned to each register.

The registers can be broken down as follows:

- At least one register must be declared as a four-component position register. This is the only vertex shader register that is required.
- The first ten registers consumed by a shader may use up to four components (xyzw) maximum.
- The last (or twelfth) register may only contain a scalar (such as point size).

For a listing of the registers, see [Registers - vs_3_0](#).

Texture Sample in a Vertex Shader

Vertex shader 3_0 supports texture lookup in the vertex shader using [texldl - vs](#).

Pixel Shader Model 3 Features

The pixel shader color and texture registers have been collapsed into ten input registers (see [Input Register Types](#)). The Face Register is a floating point scalar register. Only the sign of this register is valid. If the sign is negative the primitive is a back face. This can be

used inside a pixel shader to achieve two-sided lighting, for instance. The Position Register references the current (x,y) pixels.

The shader constant registers can be set using:

- [SetPixelShaderConstantB](#)
- [SetPixelShaderConstantI](#)
- [SetPixelShaderConstantF](#)

Match Semantics on vs_3_0 and ps_3_0 Shaders

There are some restrictions on semantic usage with vs_3_0 and ps_3_0. In general, you need to be careful when using a semantic for a shader input that matches a semantic used on a shader output.

For instance, this pixel shader packs multiple names into one register:

```
ps_3_0
dcl_texcoord0 v0.x
dcl_texcoord1 v0.yz // Valid to pack multiple names into one register
dcl_texcoord2_centroid v1.w
...
```

Each register has a different semantic. Notice that you can also name v0.x and v0.yz with different (multiple) semantics because of the use of the write mask.

Given the pixel shader, the following vs_3_0 shader cannot be paired with it:

```
vs_3_0
...
dcl_texcoord0 o5.x
dcl_texcoord1 o6.yzw
...
```

These two shaders conflict with their use of the [D3DDECLUSAGE_TEXCOORD0](#) And [D3DDECLUSAGE_TEXCOORD1](#) semantics.

Rewrite the vertex shader like this to avoid the semantic collision:

```
vs_3_0
...
dcl_texcoord2 o3
dcl_texcoord3 o9
...
```

Similarly, a semantic name declared on different input registers in the pixel shader (v0 and v1 in the pixel shader) cannot be used in a single output register in this vertex shader. For instance, this vertex shader cannot be paired with the pixel shader because D3DDECLUSAGE_TEXCOORD1 is used for both pixel shader input registers (v0, v1) and the vertex shader output register o3.

```
vs_3_0
...
dcl_texcoord0 o3.x
dcl_texcoord1 o3.yz

dcl_texcoord2 o3.w // BAD! Would be valid if this were not o3
dcl_texcoord3 o9 ...
```

On the other hand, this vertex shader cannot be paired with the pixel shader because the output mask for a parameter with a given semantic does not provide the data that is requested by the pixel shader:

```
vs_3_0
...
dcl_texcoord0 o5.x
dcl_texcoord1 o5.yzw
dcl_texcoord2 o7.yz // BAD! Would be valid if w were included
dcl_texcoord3 o9
...
```

This vertex shader does not provide an output with one of the semantic names requested by the pixel shader, so the shader pairing is invalid:

```
vs_3_0
...
dcl_texcoord0 o5.x
dcl_texcoord1 o5.yzw
dcl_texcoord3 o9
// The pixel shader wants texcoord2, with a w component,
```

```
// but it isn't output by this vertex shader at all!  
...
```

Fog, Depth, and Shading Mode Changes

When D3DRS_SHADEMODE is set for flat shading during clipping and triangle rasterization, attributes with D3DDECLUSAGE_COLOR are interpolated as flat shaded. If any components of a register are declared with a color semantic but other components of the same register are given different semantics, flat shading interpolation (linear vs. flat) will be undefined on the components in that register without a color semantic.

If fog rendering is desired, vs_3_0 and ps_3_0 shaders must implement fog. No fog calculations are done outside of the shaders. There is no fog register in vs_3_0, and additional semantics D3DDECLUSAGE_FOG (for fog blend factor computed per vertex) and D3DDECLUSAGE_DEPTH (for passing in a depth value to the pixel shader to compute the fog blend factor) have been added.

Texture stage state D3DTSS_TEXCOORDINDEX is ignored when using pixel shader 3.0.

The following values have been added to accommodate these changes:

```
// Fog and Depth usages  
D3DDECLUSAGE_FOG  
D3DDECLUSAGE_DEPTH  
  
// Additional wrap states for vs_3_0 attributes with D3DDECLUSAGE_TEXCOORD  
D3DRS_WRAP8  
D3DRS_WRAP9  
D3DRS_WRAP10  
D3DRS_WRAP11  
D3DRS_WRAP12  
D3DRS_WRAP13  
D3DRS_WRAP14  
D3DRS_WRAP15
```

Floating Point and Integer Conversions

Floating point math happens at different precision and ranges (16-bit, 24-bit, and 32-bit) in different parts of the pipeline. A value greater than the dynamic range of the pipeline that enters that pipeline (for example, a 32-bit float texture map is sampled into a 24-bit float pipeline in ps_2_0) creates an undefined result. For predictable behavior, you should clamp such a value to the dynamic range maximum.

Conversion from a floating point value to an integer happens in several places such as:

- When encountering a [mova - vs](#) instruction.
- During texture addressing.
- When writing out to a non-floating point render target.

Specifying Full or Partial Precision

Both ps_3_0 and ps_2_x provide support for two levels of precision:

ps_3_0	ps_2_0	Precision	Value
x		Full	fp32 or higher
x		Partial precision	fp16=s10e5
x	x	Full	fp24=s16e7 or higher
x	x	Partial precision	fp16=s10e5

ps_3_0 supports more precision than ps_2_0 does. By default, all operations occur at the full precision level.

Partial precision (see [Pixel Shader Register Modifiers](#)) is requested by adding the _pp modifier to shader code (provided that the underlying implementation supports it). Implementations are always free to ignore the modifier and perform the affected operations in full precision.

The _pp modifier can occur in two contexts:

- On a texture coordinate declaration to pass partial-precision texture coordinates to the pixel shader. This could be used when texture coordinates relay color data to the pixel shader, which may be faster with partial precision than with full precision in some implementations.
- On any instruction to request the use of partial precision, including texture load instructions. This indicates that the implementation is allowed to execute the instruction with partial precision and store a partial-precision result. In the absence of an explicit modifier, the instruction must be performed at full precision (regardless of the precision of the input operands).

An application might deliberately choose to trade off precision for performance. There are several kinds of shader input data which are natural candidates for partial precision processing:

- Color iterators are well represented by partial-precision values.
- Texture values from most formats can be accurately represented by partial-precision values (values sampled from 32-bit, floating-point format textures are an obvious exception).
- Constants may be represented by partial-precision representation as appropriate to the shader.

In all these cases the developer may choose to specify partial precision to process the data, knowing that no input data precision is lost. In some cases, a shader may require that the internal steps of a calculation be performed at full precision even when input and final output values do not have more than partial precision.

Software Vertex and Pixel Shaders

Software implementations (run-time and reference for vertex shaders and reference for pixel shaders) of version 2_0 shaders and above have some validation relaxed. This is useful for debugging and prototyping purposes. The application indicates to the runtime/assembler that it needs some of the validation relaxed using the _sw flag in the assembler (for example, vs_2_sw). A software shader will not work with hardware.

vs_2_sw is a relaxation to the maximum caps of vs_2_x; similarly, ps_2_sw is a relaxation to the maximum caps of ps_2_x. Specifically, the following validations are relaxed:

Shader model	Resource	Limit
vs_2_sw, vs_3_sw,	Instruction Counts	Unlimited
ps_2_sw, ps_3_sw		
vs_2_sw, vs_3_sw, ps_2_sw, ps_3_sw	Float Constant Registers	8192
vs_2_sw, vs_3_sw, ps_2_sw, ps_3_sw	Integer Constant Registers	2048
vs_2_sw, vs_3_sw, ps_2_sw, ps_3_sw	Boolean Constant Registers	2048

Shader model	Resource	Limit
ps_2_sw	Dependent-read depth	Unlimited
vs_2_sw	flow control instructions and labels	Unlimited
vs_2_sw, vs_3_sw, ps_2_sw, ps_3_sw	Loop start/step/counts	Iteration start and iteration step size for rep and loop instructions are 32-bit signed integers. Count can be up to MAX_INT/64.
vs_2_sw, vs_3_sw, ps_2_sw, ps_3_sw	Port limits	Port limits for all register files are relaxed.
vs_3_sw	Number of interpolators	16 output registers in vs_3_sw.
ps_3_sw	Number of interpolators	14(16-2) input registers for ps_3_sw.

Related topics

[Shader Model 3 \(DirectX HLSL\)](#)

Fragment Declaration Syntax (Direct3D 9 HLSL)

Article • 06/08/2021 • 2 minutes to read

Each Microsoft High Level Shader Language (HLSL) function can be converted into a shader fragment with the addition of a fragment declaration.

Syntax

```
fragmentKeyword FragmentName = compile_fragment shaderProfile  
FunctionName();
```

where:

Value	Description
fragmentKeyword	Required keyword. Either pixelfragment or vertexfragment.
FragmentName	An ASCII text string that specifies the compiled fragment name.
compile_fragment	Required keyword.
shaderProfile	The shader model to compile against. Any valid vertex shader profile (see D3DXGetVertexShaderProfile) or pixel shader profile (see D3DXGetPixelShaderProfile).
FunctionName()	The shader function name, followed by parentheses.

Shared fragment parameters are marked by adding an 'r_' prefix to their semantic.

```
void AmbientDiffuse( float3 vPosWorld: r_PosWorld,  
                      float3 vNormalWorld: r_NormalWorld,  
                      out float4 vColor: COLOR0 )  
{  
    // Compute the light vector  
    float3 vLight = normalize( g_vLightPosition - vPosWorld );  
  
    // Compute the ambient and diffuse components of illumination  
    vColor = g_vLightColor * g_vMaterialAmbient;
```

```
    vColor += g_vLightColor * g_vMaterialDiffuse * saturate( dot( vLight,
vNormalWorld ) );
}
vertexfragment AmbientDiffuseFragment = compile_fragment vs_1_1
AmbientDiffuse();
```

In this example, the r_PosWorld and r_NormalWorld semantics identify that these two parameters are shared parameters among other fragments.

 **Note**

Fragment linker was a Microsoft Direct3D 9 technology in D3DX 9. Fragment linker was a tool (Flink.exe), a D3DX 9 API, and an HLSL enhancement. Fragment linker was dropped as of the August 2009 DirectX SDK release. Fragment linker never applied to Microsoft Direct3D 10, Microsoft Direct3D 10.1, or Microsoft Direct3D 11.

Related topics

[Shader Model 3 \(DirectX HLSL\)](#)

Shader Model 4

Article • 08/19/2021 • 2 minutes to read

Shader Model 4 is a superset of the capabilities in [Shader Model 3](#), except that Shader Model 4 doesn't support the features in Shader Model 1. It has been designed using a common-shader core that gives a common set of features to all programmable shaders, which are only programmable using HLSL.

Feature	Capability
Instruction Set	HLSL functions
Register Set	The register set is accessible through members in constant and texture buffers using HLSL semantics for things like component packing. <ul style="list-style-type: none">Pixel shader registers (see Registers - ps_4_0 and Registers - ps_4_1)Vertex shader registers (see Registers - vs_4_0 and Registers - vs_4_1)Geometry shader registers (see Registers - gs_4_0 and Registers - gs_4_1)
Vertex Shader Max	No restriction
Pixel Shader Max	No restriction
New Shader Profiles Added	gs_4_0, ps_4_0, vs_4_0, gs_4_1*, ps_4_1*, gs_4_1*
New Effect-Framework Profile Added	fx_4_0, fx_4_1*

* - gs_4_1, ps_4_1, vs_4_1 and fx_4_1 are supported on Direct3D 10.1 or higher.

Shader Model 4 supports a new pipeline stage—the geometry-shader stage—which can be used to create or modify existing geometry. It also includes two new object types: a stream-output object designed for streaming data out of the geometry stage, and a templated texture object that implements texture sampling functions.

- [Common-Shader Core](#)
- [Constants](#)
- [Geometry-Shader Object](#)
- [Stream-Output Object](#)
- [Texture Object](#)

Shader Model 4 supports packing rules that dictate how tightly data can be arranged when it is stored. These rules are described in [Packing Rules for Constant Variables](#)

The [Shader Model 4 Assembly](#) section describes the assembly instructions that the Shader Model 4 and Shader Model 4.1 support.

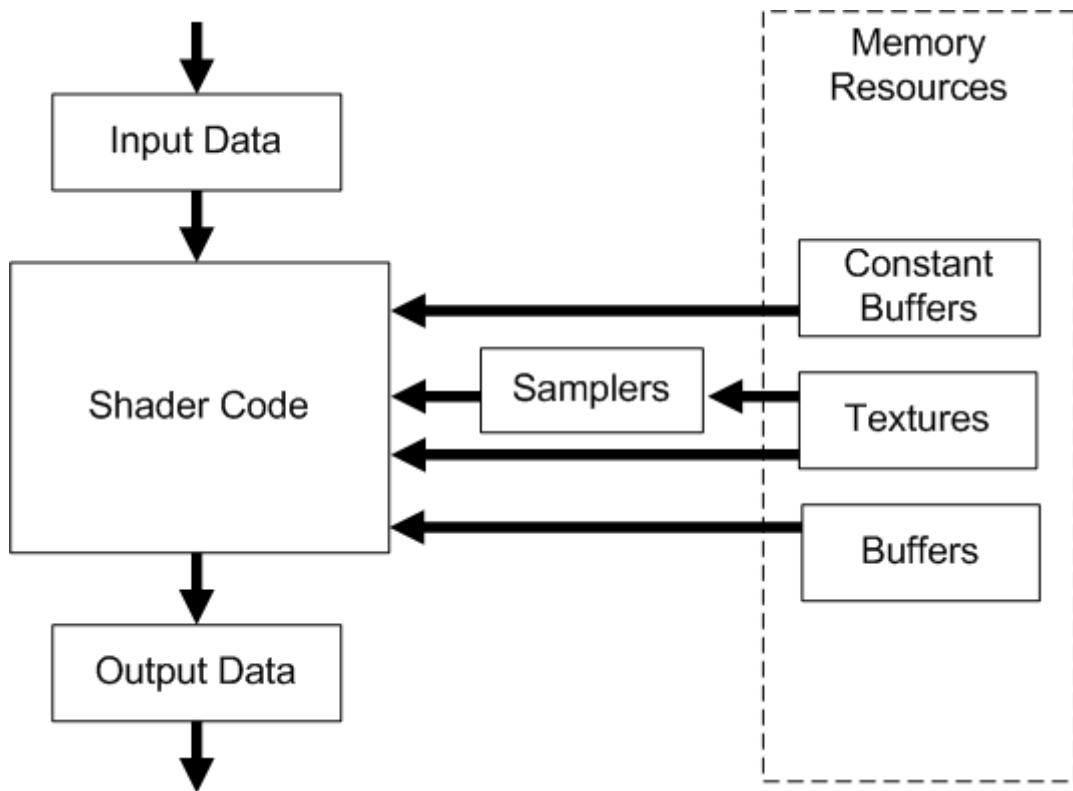
Related topics

[Shader Models vs Shader Profiles](#)

Common-Shader Core

Article • 06/30/2021 • 3 minutes to read

In Shader Model 4, all shader stages implement the same base functionality using a common-shader core. In addition, each of the three shader stages (vertex, geometry, and pixel) offer functionality unique to each stage, such as the ability to generate new primitives from the geometry shader stage or to discard a specific pixel in the pixel shader stage. The following diagram shows how data flows through a shader stage, and the relationship of the common-shader core with shader memory resources.



- **Input Data:** A vertex shader receives its inputs from the input assembler stage; geometry and pixel shaders receive their inputs from the previous shader stage. Additional inputs include [system-value semantics](#), which are consumable by the first unit in the pipeline to which they are applicable.
- **Output Data:** Shaders generate output results to be passed onto the subsequent stage in the pipeline. For a geometry shader, the amount of data output from a single invocation can vary. Some outputs are interpreted by the common-shader core (such as vertex position and render-target-array index), others are designed to be interpreted by an application.
- **Shader Code:** Shaders can read from memory, perform vector floating point and integer arithmetic operations, or flow control operations. There is no limit to the number of statements that can be implemented in a shader.
- **Samplers:** Samplers define how to sample and filter textures. As many as 16 samplers can be bound to a shader simultaneously.

- **Textures:** Textures can be filtered using samplers or read on a per-texel basis directly with the `load` intrinsic function.
- **Buffers:** Buffers are never filtered, but can be read from memory on a per-element basis directly with the `load` intrinsic function. As many as 128 texture and buffer resources (combined) can be bound to a shader simultaneously.
- **Constant Buffers:** Constant buffers are optimized for shader constant-variables. As many as 16 constant buffers can be bound to a shader stage simultaneously. They are designed for more frequent update from the CPU; therefore, they have additional size, layout, and access restrictions.

Differences between Direct3D 9 and Direct3D 10:

- In Direct3D 9, each shader unit had a single, small constant register file to store all constant shader variables. Accommodating all shaders with this limited constant space required frequent recycling of constants by the CPU.
- In Direct3D 10, constants are stored in immutable buffers in memory and are managed like any other resource. There is no limit to the number of constant buffers an application can create. By organizing constants into buffers by frequency of update and usage, the amount of bandwidth required to update constants to accommodate all shaders can be significantly reduced.

Integer and Bitwise Support

The common shader core provides a full set of IEEE-compliant 32-bit integer and bitwise operations. These operations enable a new class of algorithms in graphics hardware examples include compression and packing techniques, FFTs, and bitfield program-flow control.

The `int` and `uint` data types in Direct3D 10 HLSL map to 32-bit integers in hardware.

Differences between Direct3D 9 and Direct3D 10:

In Direct3D 9 stream inputs marked as integer in HLSL were interpreted as floating-point. In Direct3D 10, stream inputs marked as integer are interpreted as a 32-bit integer.

In addition, boolean values are now all bits set or all bits unset. Data converted to `bool` will be interpreted as true if the value is not equal to 0.0f (both positive and negative zero are allowed to be false) and false otherwise.

Bitwise operators

The common shader core supports the following bitwise operators:

Operator	Function
<code>~</code>	Logical Not
<code><<</code>	Left Shift
<code>>></code>	Right Shift
<code>&</code>	Logical And
<code> </code>	Logical Or
<code>^</code>	Logical Xor
<code><<=</code>	Left shift Equal
<code>>>=</code>	Right Shift Equal
<code>&=</code>	And Equal
<code> =</code>	Or Equal
<code>^=</code>	Xor Equal

Bitwise operators are defined to operate only on `int` and `uint` data types. Attempting to use bitwise operators on `float` or `struct` data types will result in an error. Bitwise operators follow the same precedence as C with regard to other operators.

Binary Casts

Casting between an integer and a floating-point type will convert the numeric value following C truncation rules. Casting a value from a `float`, to an `int`, and back to a `float` is a lossy conversion dependent on the precision of the target data type. Here are some of the conversion functions: [`asfloat` \(DirectX HLSL\)](#), [`asint` \(DirectX HLSL\)](#), [`asuint` \(DirectX HLSL\)](#).

Binary casts can also be performed using HLSL intrinsic functions. These cause the compiler to reinterpret the bit representation of a number into the target data type.

Related topics

[Shader Model 4](#)

Shader Constants (HLSL)

Article • 06/30/2021 • 4 minutes to read

In Shader Model 4, shader constants are stored in one or more buffer resources in memory. They can be organized into two types of buffers: constant buffers (cbuffers) and texture buffers (tbuffers). Constant buffers are optimized for constant-variable usage, which is characterized by lower-latency access and more frequent update from the CPU. For this reason, additional size, layout, and access restrictions apply to these resources. Texture buffers are accessed like textures and perform better for arbitrarily indexed data. Regardless of which type of resource you use, there is no limit to the number of constant buffers or texture buffers an application can create.

Declaring a constant buffer or a texture buffer looks very much like a structure declaration in C, with the addition of the **register** and **packoffset** keywords for manually assigning registers or packing data.

```
BufferType [Name] [: register(b#)] { VariableDeclaration [: packoffset(c#.xyzw)]; ... };
```

Parameters

BufferType

[in] The buffer type.

BufferType	Description
cbuffer	constant buffer
tbuffer	texture buffer

Name

[in] Optional, ASCII string containing a unique buffer name.

register(b#)

[in] Optional keyword, used to manually pack constant data. Constants can be packed in a register only in a constant buffer, where the starting register is given by the register number (#).

VariableDeclaration

[in] Variable declaration, similar to a structure member declaration. This can be any HLSL type or effect object (except a texture or a sampler object).

packoffset(c#.xyzw)

[in] Optional keyword, used to manually pack constant data. Constants can be packed in any constant buffer, where the register number is given by (#). Sub-component packing (using xyzw swizzling) is available for constants whose size fit within a single register (do not cross a register boundary). For instance, a float4 could not be packed in a single register starting with the y-component because it would not fit in a four-component register.

Remarks

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.

A constant buffer is a specialized buffer resource that is accessed like a buffer. Each constant buffer can hold up to 4096 [vectors](#); each vector contains up to four 32-bit values. You can bind up to 14 constant buffers per pipeline stage (2 additional slots are reserved for internal use).

A texture buffer is a specialized buffer resource that is accessed like a texture. Texture access (as compared with buffer access) can have better performance for arbitrarily indexed data. You can bind up to 128 texture buffers per pipeline stage.

A buffer resource is designed to minimize the overhead of setting shader constants. The effect framework (see [ID3D10Effect Interface](#)) will manage updating constant and texture buffers, or you can use the Direct3D API to update buffers (see [Copying and Accessing Resource Data \(Direct3D 10\)](#) for information). An application can also copy data from another buffer (such as a render target or a stream-output target) into a constant buffer.

For more info on using constant buffers in a D3D10 application, see [Resource Types \(Direct3D 10\)](#) and [Creating Buffer Resources \(Direct3D 10\)](#).

For more info on using constant buffers in a D3D11 application, see [Introduction to Buffers in Direct3D 11](#) and [How to: Create a Constant Buffer](#).

A constant buffer does not require a [view](#) to be bound to the pipeline. A texture buffer, however, requires a view and must be bound to a texture slot (or must be bound with [SetTextureBuffer](#) when using an effect).

There are two ways to pack constants data: using the [register \(DirectX HLSL\)](#) and [packoffset \(DirectX HLSL\)](#) keywords.

Differences between Direct3D 9 and Direct3D 10 and 11:

- Unlike the auto-allocation of constants in Direct3D 9, which did not perform packing and instead assigned each variable to a set of float4 registers, HLSL constant variables follow packing rules in Direct3D 10 and 11.

Organizing constant buffers

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.

The best way to efficiently use constant buffers is to organize shader variables into constant buffers based on their frequency of update. This allows an application to minimize the bandwidth required for updating shader constants. For example, a shader might declare two constant buffers and organize the data in each based on their frequency of update: data that needs to be updated on a per-object basis (like a world matrix) is grouped into a constant buffer which could be updated for each object. This is separate from data that characterizes a scene and is therefore likely to be updated much less often (when the scene changes).

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

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

Default constant buffers

There are two default constant buffers available, \$Global and \$Param. Variables that are placed in the global scope are added implicitly to the \$Global cbuffer, using the same packing method that is used for cbuffers. Uniform parameters in the parameter list of a

function appear in the \$Param constant buffer when a shader is compiled outside of the effects framework. When compiled inside the effects framework, all uniforms must resolve to variables defined in the global scope.

Examples

Here is an example from [Skinning10 Sample](#) that is a texture buffer made up of an array of matrices.

```
tbuffer tbAnimMatrices
{
    matrix g_mTexBoneWorld[MAX_BONE_MATRICES];
};
```

This example declaration manually assigns a constant buffer to start at a particular register, and also packs particular elements by subcomponents.

```
cbuffer MyBuffer : register(b3)
{
    float4 Element1 : packoffset(c0);
    float1 Element2 : packoffset(c1);
    float1 Element3 : packoffset(c1.y);
}
```

Related topics

[Shader Model 4](#)

Geometry-Shader Object

Article • 06/30/2021 • 2 minutes to read

A geometry-shader object processes entire primitives. Use the following syntax to declare a geometry-shader object.

```
[maxvertexcount(NumVerts)] void ShaderName ( PrimitiveType DataType Name [ NumElements ], inout StreamOutputObject );
```

Parameters

[maxvertexcount(*NumVerts*)]

[in] Declaration for the maximum number of vertices to create.

- [maxvertexcount()] - required keyword; brackets and parenthesis are required characters for correct syntax.
- *NumVerts* - An integer number representing the number of vertices.

ShaderName

[in] An ASCII string that contains a unique name for the geometry-shader function.

PrimitiveType *DataType* *Name* [*NumElements*]

PrimitiveType - Primitive type, which determines the order of the primitive data.

Primitive Type	Description
<i>point</i>	Point list
<i>line</i>	Line list or line strip
<i>triangle</i>	Triangle list or triangle strip
<i>lineadj</i>	Line list with adjacency or line strip with adjacency
<i>triangleadj</i>	Triangle list with adjacency or triangle strip with adjacency

DataType - [in] An input data type; can be any [HLSL data type](#).

Name - Argument name; this is an ASCII string.

NumElements - Array size of the input, which depends on the *PrimitiveType* as shown in the following table.

Primitive Type	NumElements
<i>point</i>	[1] You operate on only one point at a time.
<i>line</i>	[2] A line requires two vertices.
<i>triangle</i>	[3] A triangle requires three vertices.
<i>lineadj</i>	[4] A lineadj has two ends; therefore, it requires four vertices.
<i>triangleadj</i>	[6] A triangleadj borders three more triangles; therefore, it requires six vertices.

StreamOutputObject

The declaration of the [stream-output object](#).

Return Value

None

Remarks

The following diagram shows the various primitive types for a geometry shader object.

Geometry Shader Input Primitives

Legend:

- = Vertex



$v[n][e]$ = Input vertex n, element e (n and e independently indexable)

POINT

$v[0][e]$

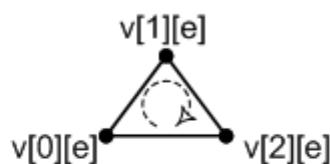
LINE

$v[0][e]$ — $v[1][e]$

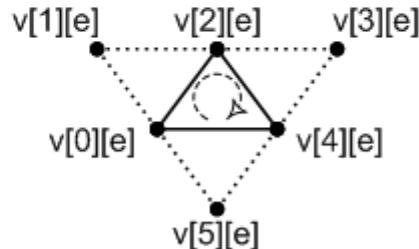
LINE_ADJ

$v[0][e]$ — $v[1][e]$ — $v[2][e]$ — $v[3][e]$

TRIANGLE



TRIANGLE_ADJ



1-32 CONTROL POINT PATCH

$v[0][e]$

•

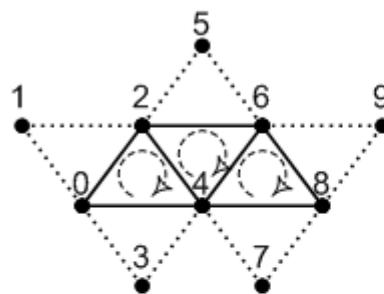
$v[n][e]$

The following diagram shows geometry shader invocations.

Additional Input
vPrim: 32bit scalar, available for all input primitives. Generated by Input Assembler as "PrimitiveID". Only single value, for the interior primitive, not any adjacent primitives.

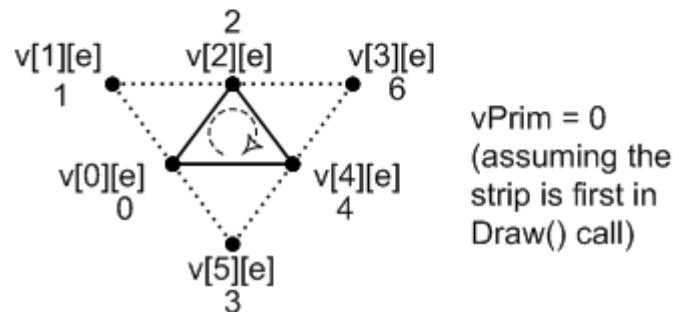
Example: GS Invocations From TriStrip w/Adjacency

Triangle strip with adjacency,
generated by Input Assembler :

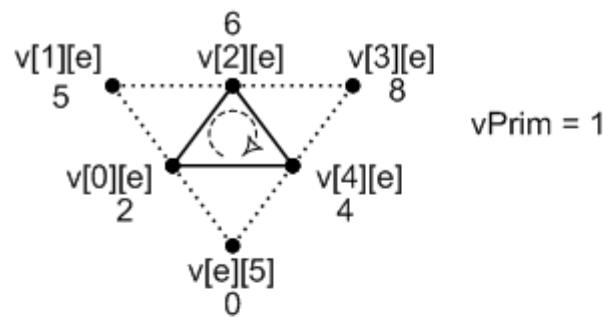


Resulting Geometry Shader
Invocations (TRIANGLE_ADJ) :

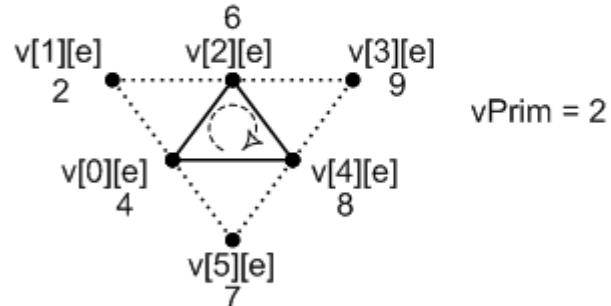
First invocation:



Second invocation:



Third invocation:



In general, for each GS invocation, $v[0][\#]$ is initialized with the leading vertex for the primitive, and other vertices are obtained by traversing around the primitive in the direction of the winding order.

Examples

This example is from exercise 1 from the [Direct3D 10 Shader Model 4.0 Workshop](#).

```
[maxvertexcount(3)]
void GSScene( triangleadj GSSceneIn input[6], inout
TriangleStream<PSSceneIn> OutputStream )
{
    PSSceneIn output = (PSSceneIn)0;

    for( uint i=0; i<6; i+=2 )
    {
        output.Pos = input[i].Pos;
        output.Norm = input[i].Norm;
        output.Tex = input[i].Tex;

        OutputStream.Append( output );
    }

    OutputStream.RestartStrip();
}
```

Minimum Shader Model

This object is supported in the following shader models.

Shader Model	Supported
Shader Model 4 and higher shader models	yes

Related topics

[Shader Model 4](#)

Stream-Output Object

Article • 08/19/2020 • 2 minutes to read

A stream-output object is a templated object that streams data out of the [geometry-shader stage](#). Use the following syntax to declare a stream-output object.

```
inout StreamOutputObject<DataType> Name;
```

Parameters

StreamOutputObject < DataType > Name

The stream-output object (SO) declaration.

Stream-Output Object Types	Description
<i>PointStream</i>	A sequence of point primitives
<i>LineStream</i>	A sequence of line primitives
<i>TriangleStream</i>	A sequence of triangle primitives

DataType - Output data type; can be any [HLSL data type](#). Must be surrounded by the angle brackets.

Name - Variable name; an ASCII string that uniquely identifies the object.

Example

This is an example of a stream-output object declaration that streams out triangle primitives whose data is defined by the PS_CUBEMAP_IN structure. The geometry-shader is limited to generating 18 vertices.

```
struct PS_CUBEMAP_IN
{
    float4 Pos : SV_POSITION;      // Projection coord
    float2 Tex : TEXCOORD0;        // Texture coord
    uint RTIndex : SV_RenderTargetArrayIndex;
```

```
};

[maxvertexcount(18)]
void main( inout TriangleStream<PS_CUBEMAP_IN> CubeMapStream, triangle
PS_CUBEMAP_INT[3] )
{
    ...
}
```

This is a code snippet from the [CubeMapGS Sample](#).

Stream-Output Object Methods

Use the following syntax to call stream-output-object methods.

```
Object.Method
```

The following methods are implemented.

Methods	Description
Append	Append output data to an existing stream.
RestartStrip	End the current primitive strip and start a new primitive strip.

Minimum Shader Model

This object is supported in the following shader models.

Shader Model	Supported
Shader Model 4 and higher shader models	yes

Related topics

[Shader Model 4](#)

Append (DirectX HLSL Stream-Output Object)

Article • 06/30/2021 • 2 minutes to read

Append geometry-shader-output data to an existing stream.

`Append(StreamDataType);`

Parameters

Item	Description
<code>StreamDataType</code>	A data input description. This description must match the stream-object template parameter called <code>DataType</code> .

Return Value

None

Example

This code snippet (from the [CubeMapGS Sample](#)) shows a partial example of appending triangle strip primitives to a stream-output object.

```
[maxvertexcount(18)]
void GS_CubeMap( triangle GS_CUBEMAP_IN input[3],
                  inout TriangleStream<PS_CUBEMAP_IN> CubeMapStream )
{
    for( int f = 0; f < 6; ++f )
    {
        // Compute screen coordinates
        PS_CUBEMAP_IN output;
        output.RTIndex = f;
        for( int v = 0; v < 3; v++ )
        {
            output.Pos = mul( input[v].Pos, g_mViewCM[f] );
            output.Pos = mul( output.Pos, mProj );
            output.Tex = input[v].Tex;
            CubeMapStream.Append( output );
        }
        CubeMapStream.RestartStrip();
    }
}
```

```
    }  
}
```

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Stream-Output Object](#)

RestartStrip (DirectX HLSL Stream-Output Object)

Article • 06/30/2021 • 2 minutes to read

Ends the current primitive strip and starts a new strip. If the current strip does not have enough vertices emitted to fill the primitive topology, the incomplete primitive at the end will be discarded.

```
RestartStrip();
```

Parameters

Item	Description
None	

Return Value

None

Remarks

A strip cut causes the current strip to end, and a new strip to start. A strip cut can be done by explicitly calling this method, or just by rendering up to the maximum index value (1, which is 0xffffffff for 32-bit indices or 0xffff for 16-bit indices). Each instance of an indexed-instanced draw generates a strip cut automatically. This is true even if the topology is not a triangle strip.

ⓘ Note

Support for restart and the 1 'magic value' for a cut is only available on **feature level 10.0 or higher** devices.

The output is always assumed to be a triangle strip. To make the output a triangle list, you must call `RestartStrip` between each triangle. Triangle fans are unsupported.

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Stream-Output Object](#)

Texture Object

Article • 08/20/2021 • 4 minutes to read

In Direct3D 10, you specify the samplers and textures independently; texture sampling is implemented by using a templated-texture object. This templated-texture object has a specific format, returns a specific type, and implements several methods.

Differences between Direct3D9 and Direct3D10:

- In Direct3D 9, samplers are bound to specific textures.
- In Direct3D 10, textures and samplers are independent objects. Each templated-texture object implements texture sampling methods that take both the texture and the sampler as input parameters.

Here is the syntax for creating all texture objects (except multisampled objects).

Object1 [<i>Type</i>] <i>Name</i>;

Multisampled objects (Texture2DMS and Texture2DMSArray) require the texture size to be explicitly stated and expressed as the number of samples.

Object2 [<i>Type, Samples</i>] <i>Name</i>;
--

Parameters

Item	Description
------	-------------

Item	Description																								
<i>Object</i>	A texture object. Must be one of the following types.																								
	<table border="1"> <thead> <tr> <th>Object1 Type</th><th>Description</th></tr> </thead> <tbody> <tr> <td>Buffer</td><td>Buffer</td></tr> <tr> <td>Texture1D</td><td>1D texture</td></tr> <tr> <td>Texture1DArray</td><td>Array of 1D textures</td></tr> <tr> <td>Texture2D</td><td>2D texture</td></tr> <tr> <td>Texture2DArray</td><td>Array of 2D textures</td></tr> <tr> <td>Texture3D</td><td>3D texture</td></tr> <tr> <td>TextureCube</td><td>Cube texture</td></tr> <tr> <td>TextureCubeArray</td><td>Array of cube textures</td></tr> <tr> <td>Object2 Type</td><td>Description</td></tr> <tr> <td>Texture2DMS</td><td>2D multisampled texture</td></tr> <tr> <td>Texture2DMSArray</td><td>Array of 2D multisampled textures</td></tr> </tbody> </table>	Object1 Type	Description	Buffer	Buffer	Texture1D	1D texture	Texture1DArray	Array of 1D textures	Texture2D	2D texture	Texture2DArray	Array of 2D textures	Texture3D	3D texture	TextureCube	Cube texture	TextureCubeArray	Array of cube textures	Object2 Type	Description	Texture2DMS	2D multisampled texture	Texture2DMSArray	Array of 2D multisampled textures
Object1 Type	Description																								
Buffer	Buffer																								
Texture1D	1D texture																								
Texture1DArray	Array of 1D textures																								
Texture2D	2D texture																								
Texture2DArray	Array of 2D textures																								
Texture3D	3D texture																								
TextureCube	Cube texture																								
TextureCubeArray	Array of cube textures																								
Object2 Type	Description																								
Texture2DMS	2D multisampled texture																								
Texture2DMSArray	Array of 2D multisampled textures																								
	<ol style="list-style-type: none"> 1. The Buffer type supports most texture object methods except GetDimensions. 2. TextureCubeArray is available in shader model 4.1 or higher. 3. Shader model 4.1 is available in Direct3D 10.1 or higher. 																								
<i>Type</i>	Optional. Any scalar HLSL type or vector HLSL type , surrounded by angle brackets. The default type is float4 .																								
<i>Name</i>	An ASCII string that specifies the texture object name.																								
<i>Samples</i>	The number of samples (ranges between 1 and 128).																								

Example 1

Here is an example of declaring a texture object.

```
Texture2D <float4> MyTex;  
Texture2DMS <float4, 128> MyMSTex;
```

Texture Object Methods

Each texture object implements certain methods; here's the table that lists all of the methods. See the reference page for each method to see what objects can use that method.

Texture Method	Description	vs_4_0	vs_4_1	ps_4_0	ps_4_1	gs_4_0	gs_4_1
CalculateLevelOfDetail	Calculate the LOD, return a clamped result.					x	
CalculateLevelOfDetailUnclamped	Calculate the LOD, return an unclamped result.					x	
Gather	Gets the four samples (red component only) that would be used for bilinear interpolation when sampling a texture.		x		x		x
GetDimensions	Get the texture dimension for a specified mipmap level.	x	x	x	x	x	x
GetDimensions (MultiSample)	Get the texture dimension for a specified mipmap level.		x		x		x

Texture Method	Description	vs_4_0	vs_4_1	ps_4_0	ps_4_1	gs_4_0	gs_4_1
GetSamplePosition	Get the position of the specified sample.		x		x		x
Load	Load data without any filtering or sampling.	x	x	x	x	x	x
Load (Multisample)	Load data without any filtering or sampling.		x	x	x		x
Sample	Sample a texture.			x	x		
SampleBias	Sample a texture, after applying the bias value to the mipmap level.			x	x		
SampleCmp	Sample a texture, using a comparison value to reject samples.			x	x		
SampleCmpLevelZero	Sample a texture (mipmap level 0 only), using a comparison value to reject samples.	x	x	x	x	x	x

Texture Method	Description	vs_4_0	vs_4_1	ps_4_0	ps_4_1	gs_4_0	gs_4_1
SampleGrad	Sample a texture using a gradient to influence the way the sample location is calculated.	x	x	x	x	x	x
SampleLevel	Sample a texture on the specified mipmap level.	x	x	x	x	x	x

Return Type

The return type of a texture object method is `float4` unless specified otherwise, with the exception of the multisampled anti-aliased texture objects that always need the type and sample count specified. The return type is the same as the texture resource type ([DXGI_FORMAT](#)). In other words, it can be any of the following types.

Type	Description
<code>float</code>	32-bit float (see Floating-Point Rules for differences from IEEE float)
<code>int</code>	32-bit signed integer
<code>unsigned int</code>	32-bit unsigned integer
<code>snorm</code>	32-bit float in range -1 to 1 inclusive (see Floating-Point Rules for differences from IEEE float)
<code>unorm</code>	32-bit float in range 0 to 1 inclusive (see Floating-Point Rules for differences from IEEE float)
any texture type or struct	The number of components returned must be between 1 and 3 inclusive.

In addition, the return type can be any texture type including a structure but, it must be less than 4 components such as a `float1` type which returns one component.

Default Values for Missing Components in a Texture

The default value for missing components in a texture resource type is zero for any component except the alpha component (A); the default value for the missing A is one. The way that this one appears to the shader depends on the texture resource type. It takes the form of the first typed component that is actually present in the texture resource type (starting from the left in RGBA order). If this form is UNORM or FLOAT, the default value for the missing A is 1.0f. If the form is SINT or UINT, the default value for the missing A is 0x1.

For example, when a shader reads the [DXGI_FORMAT_R24_UNORM_X8_TYPELESS](#) texture resource type, the default values for G and B are zero and the default value for A is 1.0f; when a shader reads the [DXGI_FORMAT_R16G16_UINT](#) texture resource type, the default value for B is zero and the default value for A is 0x00000001; when a shader reads the [DXGI_FORMAT_R16_SINT](#) texture resource type, the default values for G and B are zero and the default value for A is 0x00000001.

Example 2

Here is an example of texture sampling using a texture method.

```
sampler MySamp;
Texture2D <float4> MyTex;

float4 main( float2 TexCoords[2] : TEXCOORD ) : SV_Target
{
    return MyTex.Sample( MySamp, TexCoords[0] );
}
```

Minimum Shader Model

This object is supported in the following shader models.

Shader Model	Supported
Shader Model 4 and higher shader models	yes

See also

Shader Model 4

CalculateLevelOfDetail (DirectX HLSL Texture Object)

Article • 08/20/2021 • 2 minutes to read

Calculates the level of detail.

```
ret Object.CalculateLevelOfDetail( sampler_state S, float x );
```

Parameters

Item	Description								
Object	Any texture-object type (except Texture2DMS and Texture2DMSArray).								
S	[in] A Sampler state . This is an object declared in an effect file that contains state assignments.								
x	[in] The linear interpolation value or values, which is a floating-point number between 0.0 and 1.0 inclusive. The number of components is dependent on the texture-object type. <table border="1"><thead><tr><th>Texture-Object Type</th><th>Parameter Type</th></tr></thead><tbody><tr><td>Texture1D, Texture1DArray</td><td>float1</td></tr><tr><td>Texture2D, Texture2DArray</td><td>float2</td></tr><tr><td>Texture3D, TextureCube, TextureCubeArray</td><td>float3</td></tr></tbody></table>	Texture-Object Type	Parameter Type	Texture1D, Texture1DArray	float1	Texture2D, Texture2DArray	float2	Texture3D, TextureCube, TextureCubeArray	float3
Texture-Object Type	Parameter Type								
Texture1D, Texture1DArray	float1								
Texture2D, Texture2DArray	float2								
Texture3D, TextureCube, TextureCubeArray	float3								

Return Value

Returns the calculated LOD, a single floating-point value.

Minimum Shader Model

This function is supported in the following shader models.

vs_4_0	vs_4_1	ps_4_0	ps_4_1	gs_4_0	gs_4_1
			x		

1. TextureCubeArray is available in Shader Model 4.1 or higher.

2. Shader Model 4.1 is available in Direct3D 10.1 or higher.

Related topics

[Texture-Object](#)

CalculateLevelOfDetailUnclamped (DirectX HLSL Texture Object)

Article • 11/20/2019 • 2 minutes to read

Calculates the LOD without clamping the result.

```
ret Object.CalculateLevelOfDetailUnclamped( sampler_state S, float x );
```

This function works identically to [CalculateLevelOfDetail \(DirectX HLSL Texture Object\)](#), except that there is no clamping on the calculated LOD.

Related topics

[Texture-Object](#)

Gather (DirectX HLSL Texture Object)

Article • 05/09/2022 • 2 minutes to read

Gets the four samples (red component only) that would be used for bilinear interpolation when sampling a texture.

<Template Type>4 Object.Gather(sampler_state S, float2|3|4 Location [, int2 Offset]);

Parameters

Item	Description								
<i>Object</i>	The following texture-object types are supported: Texture2D, Texture2DArray, TextureCube, TextureCubeArray.								
<i>S</i>	[in] A Sampler state . This is an object declared in an effect file that contains state assignments.								
<i>Location</i>	[in] The texture coordinates. The argument type is dependent on the texture-object type. <table border="1"><thead><tr><th>Texture-Object Type</th><th>Parameter Type</th></tr></thead><tbody><tr><td>Texture2D</td><td>float2</td></tr><tr><td>Texture2DArray, TextureCube</td><td>float3</td></tr><tr><td>TextureCubeArray</td><td>float4</td></tr></tbody></table>	Texture-Object Type	Parameter Type	Texture2D	float2	Texture2DArray, TextureCube	float3	TextureCubeArray	float4
Texture-Object Type	Parameter Type								
Texture2D	float2								
Texture2DArray, TextureCube	float3								
TextureCubeArray	float4								
<i>Offset</i>	[in] An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. The argument type is dependent on the texture-object type. For shaders targeting Shader Model 5.0 and above, the 6 least significant bits of each offset value is honored as a signed value, yielding [-32..31] range. For previous shader model shaders, offsets need to be immediate integers between -8 and 7. <table border="1"><thead><tr><th>Texture-Object Type</th><th>Parameter Type</th></tr></thead><tbody><tr><td>Texture2D, Texture2DArray</td><td>int2</td></tr><tr><td>TextureCube, TextureCubeArray</td><td>not supported</td></tr></tbody></table>	Texture-Object Type	Parameter Type	Texture2D, Texture2DArray	int2	TextureCube, TextureCubeArray	not supported		
Texture-Object Type	Parameter Type								
Texture2D, Texture2DArray	int2								
TextureCube, TextureCubeArray	not supported								

Return Value

A four-component vector, with four components of red data, whose type is the same as the texture's template type.

Minimum Shader Model

This function is supported in the following shader models.

vs_4_0	vs_4_1	ps_4_0	ps_4_1	gs_4_0	gs_4_1
x			x		x

1. TextureCubeArray is available in Shader Model 4.1 or higher.
2. Shader Model 4.1 is available in Direct3D 10.1 or higher.

Example

```
Texture2D<int1> Tex2d;
Texture2DArray<int2> Tex2dArray;
TextureCube<int3> TexCube;
TextureCubeArray<float2> TexCubeArray;

SamplerState s;

int4 main (float4 f : SV_Position) : SV_Target
{
    int2 iOffset = int2(2,3);

    int4 i1 = Tex2d.Gather(s, f.xy);
    int4 i2 = Tex2d.Gather(s, f.xy, iOffset);

    int4 i3 = Tex2dArray.Gather(s, f.xyz);
    int4 i4 = Tex2dArray.Gather(s, f.xyz, iOffset);

    int4 i5 = TexCube.Gather(s, f.xyzw);

    float4 f6 = TexCubeArray.Gather(s, f.xyzw);

    return i1+i2+i3+i4+i5+int4(f6);
}
```

Related topics

[Texture-Object](#)

GetDimensions (DirectX HLSL Texture Object)

Article • 06/30/2021 • 2 minutes to read

Gets texture size information. The syntax block shows all the parameters that are possible in the method declaration. The table in the Remarks section shows which parameters are implemented for each texture-object type.

```
void Object.GetDimensions( UINT MipLevel, typeX Width, typeX Height, typeX Elements,  
typeX Depth, typeX NumberOfLevels, typeX NumberOfSamples );
```

typeX denotes that there are two possible types: **uint** or **float**.

Parameters

Item	Description
<i>Object</i>	Any texture-object type except a Buffer object.
<i>MipLevel</i>	[in] A zero-based index that identifies the mipmap level. If this argument is not used, the first mip level is assumed.
<i>Width</i>	[out] The texture width, in texels.
<i>Height</i>	[out] The texture height, in texels.
<i>Elements</i>	[out] The number of elements in an array.
<i>Depth</i>	[out] The texture depth, in texels.
<i>NumberOfLevels</i>	[out] The number of mipmap levels.
<i>NumberOfSamples</i>	[out] The number of samples.

Return Value

None

Overloaded Methods

This table lists all the different versions of the method; versions differs by the number of input parameters. Notice that for every method that takes integer parameters, there is

an overloaded method that takes floating-point parameters.

Texture-Object Type	Input Parameters
Texture1D	UINT MipLevel, UINT Width, UINT NumberOfLevels
Texture1D	UINT Width
Texture1D	UINT MipLevel, float Width, float NumberOfLevels
Texture1D	float Width
Texture1DArray	UINT MipLevel, UINT Width, UINT Elements, UINT NumberOfLevels
Texture1DArray	UINT Width, UINT Elements
Texture1DArray	UINT MipLevel, float Width, float Elements, float NumberOfLevels
Texture1DArray	float Width, float Elements
Texture2D	UINT MipLevel, UINT Width, UINT Height, UINT NumberOfLevels
Texture2D	UINT Width, UINT Height
Texture2D	UINT MipLevel, float Width, float Height, float NumberOfLevels
Texture2D	float Width, float Height
Texture2DArray	UINT MipLevel, UINT Width, UINT Height, UINT Elements, UINT NumberOfLevels
Texture2DArray	UINT Width, UINT Height, UINT Elements
Texture2DArray	UINT MipLevel, float Width, float Height, float Elements, float NumberOfLevels
Texture2DArray	float Width, float Height, float Elements
Texture3D	UINT MipLevel, UINT Width, UINT Height, UINT Depth, UINT NumberOfLevels
Texture3D	UINT Width, UINT Height, UINT Depth
Texture3D	UINT MipLevel, float Width, float Height, float Depth, float NumberOfLevels
Texture3D	float Width, float Height, float Depth
TextureCube	UINT MipLevel, UINT Width, UINT Height, UINT NumberOfLevels
TextureCube	UINT Width, UINT Height
TextureCube	UINT MipLevel, float Width, float Height, UINT NumberOfLevels

Texture-Object Type	Input Parameters
TextureCube	float Width, float Height
TextureCubeArray	UINT MipLevel, UINT Width, UINT Height, UINT Elements, UINT NumberOfLevels
TextureCubeArray	UINT Width, UINT Height, UINT Elements
TextureCubeArray	UINT MipLevel, float Width, float Height, float Elements, float NumberOfLevels
TextureCubeArray	float Width, float Height, float Elements
Texture2DMS	UINT Width, UINT Height, UINT Samples
Texture2DMS	float Width, float Height, float Samples
Texture2DMSArray	UINT Width, UINT Height, UINT Elements, UINT Samples
Texture2DMSArray	float Width, float Height, float Elements, float Samples

Minimum Shader Model

This function is supported in the following shader models.

vs_4_0	vs_4_1	ps_4_0	ps_4_1	gs_4_0	gs_4_1
x	x	x	x	x	x

1. Returns dimensions for the largest (zeroth) mipmap level.
2. TextureCubeArray is available in Shader Model 4.1 or higher.
3. Shader Model 4.1 is available in Direct3D 10.1 or higher.

Related topics

[Texture-Object](#)

GetSamplePosition (DirectX HLSL Texture Object)

Article • 06/30/2021 • 2 minutes to read

Gets the position of the specified sample.

```
ret Object.GetSamplePosition( int s );
```

Parameters

Item	Description
<i>Object</i>	A Texture2DMS or a Texture2DMSArray texture-object type.
<i>s</i>	[in] The zero-based sample index.

Return Value

Returns the (x,y) sample position, a two-component floating-point vector.

Minimum Shader Model

This function is supported in the following shader models.

vs_4_0	vs_4_1	ps_4_0	ps_4_1	gs_4_0	gs_4_1
x		x		x	

- Shader Model 4.1 is available in Direct3D 10.1 or higher.

Remarks

A pixel shader can be evaluated at sample frequency (run a pixel shader once per sample) or at pixel frequency (run a pixel shader once per pixel). Attach the SV_SampleIndex semantic to a pixel shader input to invoke a pixel shader at sample frequency, the input value is then used as a sample index when sampling the render target.

You can interpolate a pixel shader input in several ways. To interpolate at:

- A pixel center, don't use any semantic.
- A sample, use the `SV_SampleIndex` semantic.
- A centroid location, use the `_centroid` modifier.

Related topics

[Texture-Object](#)

Load (DirectX HLSL Texture Object)

Article • 11/21/2022 • 2 minutes to read

Reads texel data without any filtering or sampling.

```
ret Object.Load(  
    typeX Location,  
    [typeX SampleIndex, ]  
    [typeX Offset ]  
)
```

typeX denotes that there are four possible types: `int`, `int2`, `int3` or `int4`.

Parameters

Object

A [texture-object](#) type (except `TextureCube` or `TextureCubeArray`).

Location

[in] The texture coordinates; the last component specifies the mipmap level. This method uses a 0-based coordinate system and not a 0.0-1.0 UV system. The argument type is dependent on the texture-object type.

Object Type	Parameter Type
Buffer	<code>int</code>
<code>Texture1D</code> , <code>Texture2DMS</code>	<code>int2</code>
<code>Texture1DArray</code> , <code>Texture2D</code> , <code>Texture2DMSArray</code>	<code>int3</code>
<code>Texture2DArray</code> , <code>Texture3D</code>	<code>int4</code>

For example, to access a 2D texture, supply integer texel coordinates for the first two components and a mipmap level for the third component.

Note

When one or more of the coordinates in *Location* exceed the u, v, or w mipmap level dimensions of the texture, **Load** returns zero in all components. Direct3D guarantees to return zero for any resource that is accessed out of bounds.

SampleIndex

[in] A sampling index. Required for multi-sample textures. Not supported for other textures.

Texture Type	Parameter Type
Texture1D, Texture1DArray, Texture2D, Texture2DArray, Texture3D, Texture2DArray, TextureCube, TextureCubeArray	not supported
Texture2DMS, Texture2DMSArray ¹	int

Offset

[in] An optional offset applied to the texture coordinates before sampling. The offset type is dependent on the texture-object type, and needs to be static.

Texture Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray, Texture2DMS, Texture2DMSArray	int2
Texture3D	int3

① Note

SampleIndex must always be specified first with multi-sample textures.

Return Value

The return type matches the type in the *Object* declaration. For example, a Texture2D object that was declared as "Texture2d<uint4> myTexture;" has a return value of type

`uint4`.

Minimum Shader Model

This function is supported in the following shader models.

vs_4_0	vs_4_1¹	ps_4_0	ps_4_1¹	gs_4_0	gs_4_1¹
x	x	x	x	x	x

- Shader Model 4.1 is available in Direct3D 10.1 or higher.

Example

This partial code example is from the Paint.fx file in the [AdvancedParticles Sample](#).

```
// Object Declarations
Buffer<float4> g_ParticleBuffer;

// Shader body calling the intrinsic function
float4 PSPaint(PSQuadIn input) : SV_Target
{
    ...
    for( int i=g_ParticleStart; i<g_NumParticles; i+=g_ParticleStep )
    {
        ...
        // load the particle
        float4 particlePos = g_ParticleBuffer.Load( i*4 );
        float4 particleColor = g_ParticleBuffer.Load( (i*4) + 2 );
        ...
    }
    ...
}
```

Related topics

[Texture-Object](#)

Sample (DirectX HLSL Texture Object)

Article • 08/20/2021 • 2 minutes to read

Samples a texture.

```
<Template Type> Object.Sample( sampler_state S, float Location [, int Offset] );
```

Parameters

Item	Description										
<i>Object</i>	Any texture-object type (except Texture2DMS and Texture2DMSArray).										
<i>S</i>	[in] A Sampler state . This is an object declared in an effect file that contains state assignments.										
<i>Location</i>	[in] The texture coordinates. The argument type is dependent on the texture-object type. <table border="1"><thead><tr><th>Texture-Object Type</th><th>Parameter Type</th></tr></thead><tbody><tr><td>Texture1D</td><td>float</td></tr><tr><td>Texture1DArray, Texture2D</td><td>float2</td></tr><tr><td>Texture2DArray, Texture3D, TextureCube</td><td>float3</td></tr><tr><td>TextureCubeArray</td><td>float4</td></tr></tbody></table>	Texture-Object Type	Parameter Type	Texture1D	float	Texture1DArray, Texture2D	float2	Texture2DArray, Texture3D, TextureCube	float3	TextureCubeArray	float4
Texture-Object Type	Parameter Type										
Texture1D	float										
Texture1DArray, Texture2D	float2										
Texture2DArray, Texture3D, TextureCube	float3										
TextureCubeArray	float4										
<i>Offset</i>	[in] An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. The texture offsets need to be static. The argument type is dependent on the texture-object type. For more info, see Applying texture coordinate offsets . <table border="1"><thead><tr><th>Texture-Object Type</th><th>Parameter Type</th></tr></thead><tbody><tr><td>Texture1D, Texture1DArray</td><td>int</td></tr><tr><td>Texture2D, Texture2DArray</td><td>int2</td></tr><tr><td>Texture3D</td><td>int3</td></tr><tr><td>TextureCube, TextureCubeArray</td><td>not supported</td></tr></tbody></table>	Texture-Object Type	Parameter Type	Texture1D, Texture1DArray	int	Texture2D, Texture2DArray	int2	Texture3D	int3	TextureCube, TextureCubeArray	not supported
Texture-Object Type	Parameter Type										
Texture1D, Texture1DArray	int										
Texture2D, Texture2DArray	int2										
Texture3D	int3										
TextureCube, TextureCubeArray	not supported										

Return value

The texture's template type, which may be a single- or multi-component vector. The format is based on the texture's [DXGI_FORMAT](#).

Minimum Shader Model

This function is supported in the following shader models.

vs_4_0	vs_4_1	ps_4_0	ps_4_1	gs_4_0	gs_4_1
	x		x		

1. TextureCubeArray is available in Shader Model 4.1 or higher.
2. Shader Model 4.1 is available in Direct3D 10.1 or higher.

Example

This partial code example is based on the BasicHLSL11.fx file in the [BasicHLSL11 Sample](#).

```
// Object Declarations
Texture2D g_MeshTexture;           // Color texture for mesh

SamplerState MeshTextureSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = Wrap;
    AddressV = Wrap;
};

struct VS_OUTPUT
{
    float4 Position : SV_POSITION; // vertex position
    float4 Diffuse : COLOR0;      // vertex diffuse color (note that
COLOR0 is clamped from 0..1)
    float2 TextureUV : TEXCOORD0; // vertex texture coords
};

VS_OUTPUT In;

// Shader body calling the intrinsic function
...
        Output.RGBColor = g_MeshTexture.Sample(MeshTextureSampler,
In.TextureUV) * In.Diffuse;
```

Remarks

Texture sampling uses the texel position to look up a texel value. An offset can be applied to the position before lookup. The sampler state contains the sampling and filtering options. This method can be invoked within a pixel shader, but it is not supported in a vertex shader or a geometry shader.

Use an offset only at an integer mipmap level; otherwise, you may get different results depending on hardware implementation or driver settings.

Calculating texel positions

Texture coordinates are floating-point values that reference texture data, which is also known as normalized texture space. Address wrapping modes are applied in this order (texture coordinates + offsets + wrap mode) to modify texture coordinates outside the [0...1] range.

For texture arrays, an additional value in the location parameter specifies an index into a texture array. This index is treated as a scaled float value (instead of the normalized space for standard texture coordinates). The conversion to an integer index is done in the following order (float + round-to-nearest-even integer + clamp to the array range).

Applying texture coordinate offsets

The offset parameter modifies the texture coordinates, in texel space. Even though texture coordinates are normalized floating-point numbers, the offset applies an integer offset. Also note that the texture offsets need to be static.

The data format returned is determined by the texture format. For example, if the texture resource was defined with the DXGI_FORMAT_A8B8G8R8_UNORM_SRGB format, the sampling operation converts sampled texels from gamma 2.0 to 1.0, filter, and writes the result as a floating-point value in the range [0..1].

Related topics

[Texture-Object](#)

SampleBias (DirectX HLSL Texture Object)

Article • 08/20/2021 • 2 minutes to read

Samples a texture, after applying the input bias to the mipmap level.

```
<Template Type> Object.SampleBias( sampler_state S, float Location, float Bias [, int Offset] );
```

Parameters

Item	Description										
<i>Object</i>	Any texture-object type (except Texture2DMS and Texture2DMSArray).										
<i>S</i>	[in] A Sampler state . This is an object declared in an effect file that contains state assignments.										
<i>Location</i>	[in] The texture coordinates. The argument type is dependent on the texture-object type. <table border="1"><thead><tr><th>Texture-Object Type</th><th>Parameter Type</th></tr></thead><tbody><tr><td>Texture1D</td><td>float</td></tr><tr><td>Texture1DArray, Texture2D</td><td>float2</td></tr><tr><td>Texture2DArray, Texture3D, TextureCube</td><td>float3</td></tr><tr><td>TextureCubeArray</td><td>float4</td></tr></tbody></table>	Texture-Object Type	Parameter Type	Texture1D	float	Texture1DArray, Texture2D	float2	Texture2DArray, Texture3D, TextureCube	float3	TextureCubeArray	float4
Texture-Object Type	Parameter Type										
Texture1D	float										
Texture1DArray, Texture2D	float2										
Texture2DArray, Texture3D, TextureCube	float3										
TextureCubeArray	float4										
<i>Bias</i>	[in] The bias value, which is a floating-point number between -16.0 and 15.99, is applied to a mip level before sampling.										

Item	Description										
<i>Offset</i>	[in] An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. The texture offsets need to be static. The argument type is dependent on the texture-object type. For more info, see Applying texture coordinate offsets .										
	<table border="1"> <thead> <tr> <th>Texture-Object Type</th><th>Parameter Type</th></tr> </thead> <tbody> <tr> <td>Texture1D, Texture1DArray</td><td>int</td></tr> <tr> <td>Texture2D, Texture2DArray</td><td>int2</td></tr> <tr> <td>Texture3D</td><td>int3</td></tr> <tr> <td>TextureCube, TextureCubeArray</td><td>not supported</td></tr> </tbody> </table>	Texture-Object Type	Parameter Type	Texture1D, Texture1DArray	int	Texture2D, Texture2DArray	int2	Texture3D	int3	TextureCube, TextureCubeArray	not supported
Texture-Object Type	Parameter Type										
Texture1D, Texture1DArray	int										
Texture2D, Texture2DArray	int2										
Texture3D	int3										
TextureCube, TextureCubeArray	not supported										

Return Value

The texture's template type, which may be a single- or multi-component vector. The format is based on the texture's [DXGI_FORMAT](#).

Minimum Shader Model

This function is supported in the following shader models.

vs_4_0	vs_4_1	ps_4_0	ps_4_1	gs_4_0	gs_4_1
	x		x		

1. TextureCubeArray is available in Shader Model 4.1 or higher.
2. Shader Model 4.1 is available in Direct3D 10.1 or higher.

Related topics

[Texture-Object](#)

SampleCmp (DirectX HLSL Texture Object)

Article • 11/17/2020 • 2 minutes to read

Samples a texture and compares a single component against the specified comparison value.

```
float Object.SampleCmp(  
    SamplerComparisonState S,  
    float Location,  
    float CompareValue,  
    [int Offset]  
);
```

The comparison is a single-component comparison between the first component stored in the texture, and the comparison value passed into the method.

This method can be invoked only from a pixel shader; it isn't supported in a vertex or geometry shader.

Parameters

Object

Any [texture-object](#) type (except Texture2DMS, Texture2DMSArray, or Texture3D).

S

[in] A sampler-comparison state, which is the sampler state plus a comparison state (a comparison function and a comparison filter). See the [sampler type](#) for details and an example.

Location

[in] The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2

Texture-Object Type	Parameter Type
Texture2DArray ¹ , TextureCube	float3
TextureCubeArray ¹	float4

CompareValue

A floating-point value to use as a comparison value.

Offset

[in] An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. The texture offsets need to be static. The argument type is dependent on the texture-object type. For more info, see [Applying texture coordinate offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray ¹	int2
TextureCube, TextureCubeArray ¹	Not supported

Return Value

Returns a floating-point value in the range [0..1].

For each texel fetched (based on the sampler configuration of the filter mode), **SampleCmp** performs a comparison of the z value (3rd component of input) from the shader against the texel value (1 if the comparison passes; otherwise 0). **SampleCmp** then blends these 0 and 1 results for each texel together as in normal texture filtering (not an average) and returns the resulting [0..1] value to the shader.

Remarks

Comparison filtering provides a basic filtering operation that is useful for percentage-closer-depth filtering.

When using this method on a floating-point resource (Instead of a signed-normalized or unsigned-normalized format), the comparison value is not automatically clamped between 0.0 and 1.0. Therefore, a manual clamp of the comparison value may be necessary for common shadowing techniques.

Use an offset only at an integer miplevel; otherwise, you may get different results depending on hardware implementation or driver settings.

Minimum Shader Model

This function is supported in the following shader models.

vs_4_0	vs_4_1²	ps_4_0	ps_4_1²	gs_4_0	gs_4_1²
	x ¹		x		

1. Texture2DArray and TextureCubeArray are available in Shader Model 4.1 or higher.
2. Shader Model 4.1 is available in Direct3D 10.1 or higher.

 **Note**

SampleCmp is also available in ps_4_0_level_9_1 and 4_0_level_9_3 when you use the techniques that are described in [Implementing shadow buffers for Direct3D feature level 9](#).

Related topics

[Texture-Object](#)

SampleCmpLevelZero (DirectX HLSL Texture Object)

Article • 11/20/2019 • 2 minutes to read

Samples a texture and compares the result to a comparison value. This function is identical to calling [SampleCmp](#) on mipmap level 0 only.

Related topics

[Texture-Object](#)

SampleGrad (DirectX HLSL Texture Object)

Article • 08/20/2021 • 2 minutes to read

Samples a texture using a gradient to influence the way the sample location is calculated.

```
<Template Type> Object.SampleGrad( sampler_state S, float Location, float DDX, float DDY [, int Offset] );
```

Parameters

Item	Description												
<i>Object</i>	Any texture-object type (except Texture2DMS and Texture2DMSArray).												
<i>S</i>	[in] A Sampler state . This is an object declared in an effect file that contains state assignments.												
<i>Location</i>	[in] The Texture coordinates. The argument type is dependent on the texture-object type. <table><thead><tr><th>Texture-Object Type</th><th>Parameter Type</th></tr></thead><tbody><tr><td>Texture1D</td><td>float</td></tr><tr><td>Texture1DArray, Texture2D</td><td>float2</td></tr><tr><td>Texture2DArray, Texture3D, TextureCube</td><td>float3</td></tr><tr><td>TextureCubeArray</td><td>float4</td></tr><tr><td>Texture2DMS, Texture2DMSArray</td><td>not supported</td></tr></tbody></table>	Texture-Object Type	Parameter Type	Texture1D	float	Texture1DArray, Texture2D	float2	Texture2DArray, Texture3D, TextureCube	float3	TextureCubeArray	float4	Texture2DMS, Texture2DMSArray	not supported
Texture-Object Type	Parameter Type												
Texture1D	float												
Texture1DArray, Texture2D	float2												
Texture2DArray, Texture3D, TextureCube	float3												
TextureCubeArray	float4												
Texture2DMS, Texture2DMSArray	not supported												

Item	Description												
<i>DDX</i>	[in] The rate of change of the surface geometry in the x direction. The argument type is dependent on the texture-object type.												
	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: left; padding: 2px;">Texture-Object Type</th><th style="text-align: left; padding: 2px;">Parameter Type</th></tr> </thead> <tbody> <tr> <td style="padding: 2px;">Texture1D, Texture1DArray</td><td style="padding: 2px;">float</td></tr> <tr> <td style="padding: 2px;">Texture2D, Texture2DArray</td><td style="padding: 2px;">float2</td></tr> <tr> <td style="padding: 2px;">Texture3D, TextureCube, TextureCubeArray</td><td style="padding: 2px;">float3</td></tr> <tr> <td style="padding: 2px;">Texture2DMS, Texture2DMSArray</td><td style="padding: 2px;">not supported</td></tr> </tbody> </table>	Texture-Object Type	Parameter Type	Texture1D, Texture1DArray	float	Texture2D, Texture2DArray	float2	Texture3D, TextureCube, TextureCubeArray	float3	Texture2DMS, Texture2DMSArray	not supported		
Texture-Object Type	Parameter Type												
Texture1D, Texture1DArray	float												
Texture2D, Texture2DArray	float2												
Texture3D, TextureCube, TextureCubeArray	float3												
Texture2DMS, Texture2DMSArray	not supported												
<i>DDY</i>	[in] The rate of change of the surface geometry in the y direction. The argument type is dependent on the texture-object type.												
	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: left; padding: 2px;">Texture-Object Type</th><th style="text-align: left; padding: 2px;">Parameter Type</th></tr> </thead> <tbody> <tr> <td style="padding: 2px;">Texture1D, Texture1DArray</td><td style="padding: 2px;">float</td></tr> <tr> <td style="padding: 2px;">Texture2D, Texture2DArray</td><td style="padding: 2px;">float2</td></tr> <tr> <td style="padding: 2px;">Texture3D, TextureCube, TextureCubeArray</td><td style="padding: 2px;">float3</td></tr> <tr> <td style="padding: 2px;">Texture2DMS, Texture2DMSArray</td><td style="padding: 2px;">not supported</td></tr> </tbody> </table>	Texture-Object Type	Parameter Type	Texture1D, Texture1DArray	float	Texture2D, Texture2DArray	float2	Texture3D, TextureCube, TextureCubeArray	float3	Texture2DMS, Texture2DMSArray	not supported		
Texture-Object Type	Parameter Type												
Texture1D, Texture1DArray	float												
Texture2D, Texture2DArray	float2												
Texture3D, TextureCube, TextureCubeArray	float3												
Texture2DMS, Texture2DMSArray	not supported												
<i>Offset</i>	[in] An optional texture coordinate offset, which can be used for any texture-object types. The offset is applied to the location before sampling. Use an offset only at an integer mipmap; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see Applying Integer Offsets .												
	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: left; padding: 2px;">Texture-Object Type</th><th style="text-align: left; padding: 2px;">Parameter Type</th></tr> </thead> <tbody> <tr> <td style="padding: 2px;">Texture1D, Texture1DArray</td><td style="padding: 2px;">int</td></tr> <tr> <td style="padding: 2px;">Texture2D, Texture2DArray</td><td style="padding: 2px;">int2</td></tr> <tr> <td style="padding: 2px;">Texture3D</td><td style="padding: 2px;">int3</td></tr> <tr> <td style="padding: 2px;">TextureCube, TextureCubeArray</td><td style="padding: 2px;">not supported</td></tr> <tr> <td style="padding: 2px;">Texture2DMS, Texture2DMSArray</td><td style="padding: 2px;">not supported</td></tr> </tbody> </table>	Texture-Object Type	Parameter Type	Texture1D, Texture1DArray	int	Texture2D, Texture2DArray	int2	Texture3D	int3	TextureCube, TextureCubeArray	not supported	Texture2DMS, Texture2DMSArray	not supported
Texture-Object Type	Parameter Type												
Texture1D, Texture1DArray	int												
Texture2D, Texture2DArray	int2												
Texture3D	int3												
TextureCube, TextureCubeArray	not supported												
Texture2DMS, Texture2DMSArray	not supported												

Return Value

The texture's template type, which may be a single- or multi-component vector. The format is based on the texture's [DXGI_FORMAT](#).

Minimum Shader Model

This function is supported in the following shader models.

vs_4_0	vs_4_1	ps_4_0	ps_4_1	gs_4_0	gs_4_1
x	x	x	x	x	x

1. TextureCubeArray is available in Shader Model 4.1 or higher.
2. Shader Model 4.1 is available in Direct3D 10.1 or higher.

Example

This partial code example is from the MotionBlur.fx file in the [MotionBlur10 Sample](#).

```
// Object Declarations
Texture2D g_txDiffuse;

SamplerState g_samLinear
{
    Filter = ANISOTROPIC;
    MaxAnisotropy = 8;
    AddressU = Wrap;
    AddressV = Wrap;
};

struct VSSceneOut
{
    float4 Pos : SV_POSITION;
    float4 Color : COLOR0;
    float2 Tex : TEXCOORD;
    float2 Aniso : ANISOTROPY;
};

float4 PSSceneMain( VSSceneOut Input ) : SV_TARGET
{
    float2 ddx = Input.Aniso;
    float2 ddy = Input.Aniso;

    // Shader body calling the intrinsic function
    float4 diff = g_txDiffuse.SampleGrad( g_samLinear, Input.Tex, ddx, ddy);
```

}

...

Related topics

[Texture-Object](#)

SampleLevel (DirectX HLSL Texture Object)

Article • 08/20/2021 • 2 minutes to read

Samples a texture using a mipmap-level offset.

```
<Template Type> Object.SampleLevel( sampler_state S, float Location, float LOD [, int Offset] );
```

This function is similar to [Sample](#) except that it uses the LOD level (in the last component of the location parameter) to choose the mipmap level. For example, a 2D texture uses the first two components for uv coordinates and the third component for the mipmap level.

Parameters

Item	Description										
<i>Object</i>	Any texture-object type (except Texture2DMS and Texture2DMSArray).										
<i>S</i>	[in] A Sampler state . This is an object declared in an effect file that contains state assignments.										
<i>Location</i>	[in] The texture coordinates. The argument type is dependent on the texture-object type. <table border="1"><thead><tr><th>Texture-Object Type</th><th>Parameter Type</th></tr></thead><tbody><tr><td>Texture1D</td><td>float</td></tr><tr><td>Texture1DArray, Texture2D</td><td>float2</td></tr><tr><td>Texture2DArray, Texture3D, TextureCube</td><td>float3</td></tr><tr><td>TextureCubeArray</td><td>float4</td></tr></tbody></table>	Texture-Object Type	Parameter Type	Texture1D	float	Texture1DArray, Texture2D	float2	Texture2DArray, Texture3D, TextureCube	float3	TextureCubeArray	float4
Texture-Object Type	Parameter Type										
Texture1D	float										
Texture1DArray, Texture2D	float2										
Texture2DArray, Texture3D, TextureCube	float3										
TextureCubeArray	float4										
	If the texture object is an array, the last component is the array index.										
<i>LOD</i>	[in] A number that specifies the mipmap level. If the value is = 0, the zero'th (biggest map) is used. The fractional value (if supplied) is used to interpolate between two mipmap levels.										

Item	Description										
<i>Offset</i>	[in] An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. The texture offsets need to be static. The argument type is dependent on the texture-object type. For more info, see Applying texture coordinate offsets .										
	<table border="1"> <thead> <tr> <th>Texture-Object Type</th><th>Parameter Type</th></tr> </thead> <tbody> <tr> <td>Texture1D, Texture1DArray</td><td>int</td></tr> <tr> <td>Texture2D, Texture2DArray</td><td>int2</td></tr> <tr> <td>Texture3D</td><td>int3</td></tr> <tr> <td>TextureCube, TextureCubeArray</td><td>not supported</td></tr> </tbody> </table>	Texture-Object Type	Parameter Type	Texture1D, Texture1DArray	int	Texture2D, Texture2DArray	int2	Texture3D	int3	TextureCube, TextureCubeArray	not supported
Texture-Object Type	Parameter Type										
Texture1D, Texture1DArray	int										
Texture2D, Texture2DArray	int2										
Texture3D	int3										
TextureCube, TextureCubeArray	not supported										

Return Value

The texture's template type, which may be a single- or multi-component vector. The format is based on the texture's [DXGI_FORMAT](#).

Minimum Shader Model

This function is supported in the following shader models.

vs_4_0	vs_4_1	ps_4_0	ps_4_1	gs_4_0	gs_4_1
x	x	x	x	x	x

1. TextureCubeArray is available in Shader Model 4.1 or higher.
2. Shader Model 4.1 is available in Direct3D 10.1 or higher.

Example

This partial code example is from the Instancing.fx file in the [Instancing10 Sample](#).

```
// Object Declarations
Texture1D g_txRandom;

SamplerState g_samPoint
{
```

```
Filter = MIN_MAG_MIP_POINT;
AddressU = Wrap;
AddressV = Wrap;
};

// Shader body calling the intrinsic function
float3 RandomDir(float fOffset)
{
    float tCoord = (fOffset) / 300.0;
    return g_txRandom.SampleLevel( g_samPoint, tCoord, 0 );
...
}
```

Related topics

[Texture-Object](#)

Packing Rules for Constant Variables

Article • 08/12/2020 • 2 minutes to read

Packing rules dictate how tightly data can be arranged when it is stored. HLSL implements packing rules for VS output data, GS input and output data, and PS input and output data. (Data is not packed for VS inputs because the IA stage cannot unpack data.)

HLSL packing rules are similar to performing a `#pragma pack 4` with Visual Studio, which packs data into 4-byte boundaries. Additionally, HLSL packs data so that it does not cross a 16-byte boundary. Variables are packed into a given four-component vector until the variable will straddle a 4-vector boundary; the next variables will be bounced to the next four-component vector.

Each structure forces the next variable to start on the next four-component vector. This sometimes generates padding for arrays of structures. The resulting size of any structure will always be evenly divisible by `sizeof(four-component vector)`.

Arrays are not packed in HLSL by default. To avoid forcing the shader to take on ALU overhead for offset computations, every element in an array is stored in a four-component vector. Note that you can achieve packing for arrays (and incur the addressing calculations) by using casting.

Following are examples of structures and their corresponding packed sizes (given: a `float1` occupies 4 bytes):

```
// 2 x 16byte elements
cbuffer IE
{
    float4 Val1;
    float2 Val2; // starts a new vector
    float2 Val3;
};

// 3 x 16byte elements
cbuffer IE
{
    float2 Val1;
    float4 Val2; // starts a new vector
    float2 Val3; // starts a new vector
};

// 1 x 16byte elements
cbuffer IE
```

```
{  
    float1 Val1;  
    float1 Val2;  
    float2 Val3;  
};  
  
// 1 x 16byte elements  
cbuffer IE  
{  
    float1 Val1;  
    float2 Val2;  
    float1 Val3;  
};  
  
// 2 x 16byte elements  
cbuffer IE  
{  
    float1 Val1;  
    float1 Val1;  
    float1 Val1;  
    float2 Val2;    // starts a new vector  
};  
  
// 1 x 16byte elements  
cbuffer IE  
{  
    float3 Val1;  
    float1 Val2;  
};  
  
// 1 x 16byte elements  
cbuffer IE  
{  
    float1 Val1;  
    float3 Val2;  
};  
  
// 2 x 16byte elements  
cbuffer IE  
{  
    float1 Val1;  
    float1 Val1;  
    float3 Val2;    // starts a new vector  
};  
  
// 3 x 16byte elements  
cbuffer IE  
{  
    float1 Val1;  
  
    struct {  
        float4 SVal1;    // starts a new vector  
        float1 SVal2;    // starts a new vector  
    };  
};
```

```

    } Val2;

};

// 3 x 16byte elements
cbuffer IE
{
    float1 Val1;
    struct {
        float1 SVal1;      // starts a new vector
        float4 SVal2;      // starts a new vector
    } Val2;
};

// 3 x 16byte elements
cbuffer IE
{
    struct {
        float4 SVal1;
        float1 SVal2;      // starts a new vector
    } Val1;

    float1 Val2;
};

```

More Aggressive Packing

You could pack an array more aggressively. For instance, given an array of float variables:

```
float4 array[16];
```

You could choose to pack it like this, without any spaces in the array:

```
static float2 aggressivePackArray[32] = (float2[32])array;
```

The tighter packing is a trade off versus the need for additional shader instructions for address computation.

Related topics

[Shader Model 4](#)

Registers - ps_4_0

Article • 08/23/2019 • 2 minutes to read

This section contains reference information for the input and output registers implemented by pixel shader version 4_0.

Input Registers

Register	Name	Count	R/W	Dimension	Indexable by r#	Defaults	Requires DCL
r#	4096(r#+x#[n])	4096(r#+x#[n])	R/W	4	No	None	Yes
x#[n]	4096(r#+x#[n])	4096(r#+x#[n])	R/W	4	Yes	None	Yes
v#	32	R	4		Yes	None	Yes
t#	128	R	1		No	None	Yes
s#	16	R	1		No	None	Yes
cb#[index]	15	R	4		Yes(Contents)	None	Yes
icb[index]	1	R	4		Yes(Contents)	None	Yes

Output Registers

Register	Name	Count	R/W	Dimension	Indexable by r#	Defaults	Requires DCL
NULL	Discard Result	N/A	W	N/A	N/A	N/A	No
o#	Output Register	8	W	N/A	N/A	4	No
oDepth	Output Depth	1	W	N/A	N/A	1	N/A

Related topics

[Shader Model 4](#)

Registers - ps_4_1

Article • 08/23/2019 • 2 minutes to read

This section contains reference information for the input and output registers implemented by pixel shader version 4_1.

Input Registers

Register	Name	Count	R/W	Dimension	Indexable by r#	Defaults	Requires DCL
r#	4096(r#+x#[n])	4096(r#+x#[n])	R/W	4	No	None	Yes
x#[n]	4096(r#+x#[n])	4096(r#+x#[n])	R/W	4	Yes	None	Yes
v#	32	R	4		Yes	None	Yes
t#	128	R	1		No	None	Yes
s#	16	R	1		No	None	Yes
cb#[index]	15	R	4		Yes(Contents)	None	Yes
icb[index]	1	R	4		Yes(Contents)	None	Yes

Output Registers

Register	Name	Count	R/W	Dimension	Indexable by r#	Defaults	Requires DCL
NULL	Discard Result	N/A	W	N/A	N/A	N/A	No
o#	Output Register	8	W	N/A	N/A	4	No
oDepth	Output Depth	1	W	N/A	N/A	1	N/A
oMask	Output MSAA Mask	1	W	N/A	N/A	1	N/A

Related topics

[Shader Model 4](#)

Registers - vs_4_0

Article • 08/23/2019 • 2 minutes to read

This section contains reference information for the input and output registers implemented by vertex shader version 4_0.

Input Registers

Register	Name	Count	R/W	Dimension	Indexable by r#	Defaults	Requires DCL
r#	4096(r#+x#[n])	4096(r#+x#[n])	R/W	4	No	None	Yes
x#[n]	4096(r#+x#[n])	4096(r#+x#[n])	R/W	4	Yes	None	Yes
v#	16	R	4		Yes	None	Yes
t#	128	R	1		No	None	Yes
s#	16	R	1		No	None	Yes
cb#[index]	15	R	4		Yes(Contents)	None	Yes
icb[index]	1	R	4		Yes(Contents)	None	Yes

Output Registers

Register	Name	Count	R/W	Dimension	Indexable by r#	Defaults	Requires DCL
NULL	Discard Result	N/A	W	N/A	N/A	N/A	No
o#	Output Register	16	W	N/A	N/A	4	Yes

Related topics

Shader Model 4

Registers - vs_4_1

Article • 08/23/2019 • 2 minutes to read

This section contains reference information for the input and output registers implemented by vertex shader version 4_1.

Input Registers

Register	Name	Count	R/W	Dimension	Indexable by r#	Defaults	Requires DCL
r#	4096(r#+x#[n])	4096(r#+x#[n])	R/W	4	No	None	Yes
x#[n]	4096(r#+x#[n])	4096(r#+x#[n])	R/W	4	Yes	None	Yes
v#	32	R	4		Yes	None	Yes
t#	128	R	1		No	None	Yes
s#	16	R	1		No	None	Yes
cb#[index]	15	R	4		Yes(Contents)	None	Yes
icb[index]	1	R	4		Yes(Contents)	None	Yes

Output Registers

Register	Name	Count	R/W	Dimension	Indexable by r#	Defaults	Requires DCL
NULL	Discard Result	N/A	W	N/A	N/A	N/A	No
o#	Output Register	32	W	N/A	N/A	4	Yes

Related topics

Shader Model 4

Registers - gs_4_0

Article • 08/23/2019 • 2 minutes to read

This section contains reference information for the input and output registers implemented by geometry shader version 4_0.

Input Registers

Register	Name	Count	R/W	Dimension	Indexable by r#	Defaults	Requires DCL
r#	4096(r#+x#[n])	4096(r#+x#[n])	R/W	4	No	None	Yes
x#[n]	4096(r#+x#[n])	4096(r#+x#[n])	R/W	4	Yes	None	Yes
v# [vertex] [element]	32	R	4(comp)*6(vert)	Yes	None	None	Yes
vprim	1	R	1	No	None	None	Yes
t#	128	R	1	No	None	None	Yes
s#	16	R	1	No	None	None	Yes
cb#[index]	15	R	4	Yes(Contents)	None	None	Yes
icb[index]	1	R	4	Yes(Contents)	None	None	Yes

Output Registers

Register	Name	Count	R/W	Dimension	Indexable by r#	Defaults	Requires DCL
NULL	Discard Result	N/A	W	N/A	N/A	N/A	No
o#	Output Register	32	W	N/A	N/A	4	Yes

Related topics

[Shader Model 4](#)

Registers - gs_4_1

Article • 08/23/2019 • 2 minutes to read

This section contains reference information for the input and output registers implemented by geometry shader versions 4_0 and 4_1.

Input Registers

Register	Name	Count	R/W	Dimension	Indexable by r#	Defaults	Requires DCL
r#	4096(r#+x#[n])	4096(r#+x#[n])	R/W	4	No	None	Yes
x#[n]	4096(r#+x#[n])	4096(r#+x#[n])	R/W	4	Yes	None	Yes
v# [vertex] [element]	32	R	4(comp)*6(vert)	Yes	None	None	Yes
vprim	1	R	1	No	None	None	Yes
t#	128	R	1	No	None	None	Yes
s#	16	R	1	No	None	None	Yes
cb#[index]	15	R	4	Yes(Contents)	None	None	Yes
icb[index]	1	R	4	Yes(Contents)	None	None	Yes

Output Registers

Register	Name	Count	R/W	Dimension	Indexable by r#	Defaults	Requires DCL
NULL	Discard Result	N/A	W	N/A	N/A	N/A	No
o#	Output Register	32	W	N/A	N/A	4	Yes

Related topics

[Shader Model 4](#)

Shader Model 5

Article • 08/19/2020 • 2 minutes to read

This section contains the reference pages for HLSL Shader Model 5.

Shader Model 5 is a superset of the capabilities in [Shader Model 4](#). It has been designed using a common-shader core which provides a common set of features to all programmable shaders, which are only programmable using HLSL.

Feature	Capability
Instruction Set	HLSL intrinsic functions
Vertex Shader Max	No restriction
Pixel Shader Max	No restriction
New Shader Profiles Added	cs_4_0, gs_4_0*, ps_4_0*, vs_4_0*, cs_4_1, gs_4_1*, ps_4_1*, vs_4_1*, cs_5_0, ds_5_0, gs_5_0, hs_5_0, ps_5_0, vs_5_0

* - gs_4_0, gs_4_1, ps_4_0, ps_4_1, vs_4_0 and vs_4_1 were introduced in Shader Model 4.0, however, DirectX 11 adds support for [structured buffers](#) and byte address buffers to Shader Model 4 running on DirectX 10 hardware.

Shader Model 5 introduces the [compute shader](#) which provides high-speed general purpose computing.

A more complete listing of Shader Model 5 features is included in a listing of the [Direct3D 11 features](#).

The [Shader Model 5 Assembly](#) section describes the assembly instructions that the Shader Model 5 supports.

In This Section

Item	Description
Shader Model 5 Attributes	Reference pages for Shader Model 5 attributes.
Shader Model 5 Intrinsic Functions	Reference pages for Shader Model 5 intrinsic functions.
Shader Model 5 Objects	Reference pages for Shader Model 5 objects and methods.
Shader Model 5 System Values	Reference pages for Shader Model 5 system values.

Related topics

[Shader Models vs Shader Profiles](#)

Shader Model 5 Attributes

Article • 11/20/2019 • 2 minutes to read

Shader Model 5 implements the attributes from Shader Model 4 and below (see the DirectX SDK), as well as the following new attributes:

- [domain](#)
- [earlydepthstencil](#)
- [instance](#)
- [maxtessfactor](#)
- [numthreads](#)
- [outputcontrolpoints](#)
- [outputtopology](#)
- [partitioning](#)
- [patchconstantfunc](#)

All of these attributes are required (except for the `instance` attribute). Failure to use them will cause compile errors.

Related topics

[Shader Model 5](#)

domain

Article • 11/20/2019 • 2 minutes to read

Defines the patch type used in the HS.

```
domain(X)
```

Remarks

X is either **tri**, **quad**, or **isoline**.

This attribute is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
	x	x			

Related topics

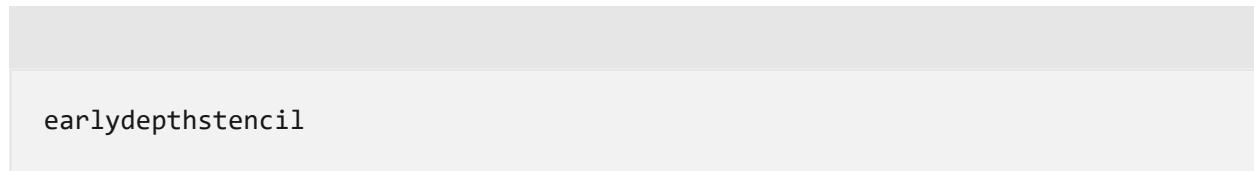
[Shader Model 5 Attributes](#)

[Shader Model 5](#)

earlydepthstencil

Article • 11/20/2019 • 2 minutes to read

Forces depth-stencil testing before a shader executes.



Remarks

Depth-stencil testing is normally done during pixel processing by a pixel shader. Forcing early depth-stencil testing causes the testing to be done prior to the shader execution; the purpose is to improve per-pixel performance by culling/reducing/avoiding unnecessary pixel processing.

An application can force early depth-stencil testing by supplying the attribute or the hardware may execute early depth-stencil testing provided no depth values are written and no unordered access operations are performed in a shader as an optimization.

This attribute is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	

Related topics

[Shader Model 5 Attributes](#)

[Shader Model 5](#)

instance

Article • 11/20/2019 • 2 minutes to read

Use this attribute to instance a geometry shader.

```
instance(x)
```

Remarks

X is an integer index that indicates the number of instances to be executed for each drawn item (for example, for each triangle). When using this attribute, use [SV_GSInstanceID](#) to identify which instance of a geometry shader is being executed.

This attribute is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
			x		

Related topics

[Shader Model 5 Attributes](#)

[Shader Model 5](#)

maxtessfactor

Article • 11/20/2019 • 2 minutes to read

Indicates the maximum value that the hull shader would return for any tessellation factor.

```
maxtessfactor(X)
```

Remarks

This attribute puts an upper bound on the amount of tessellation requested to help a driver determine the maximum amount of resources required for tessellation.

This attribute is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
	x				

Related topics

[Shader Model 5 Attributes](#)

[Shader Model 5](#)

numthreads

Article • 08/19/2020 • 2 minutes to read

Defines the number of threads to be executed in a single thread group when a compute shader is dispatched (see [ID3D11DeviceContext::Dispatch](#)).

```
numthreads(X, Y, Z)
```

The X, Y and Z values indicate the size of the thread group in a particular direction and the total of $X \times Y \times Z$ gives the number of threads in the group. The ability to specify the size of the thread group across three dimensions allows individual threads to be accessed in a manner that logically 2D and 3D data structures.

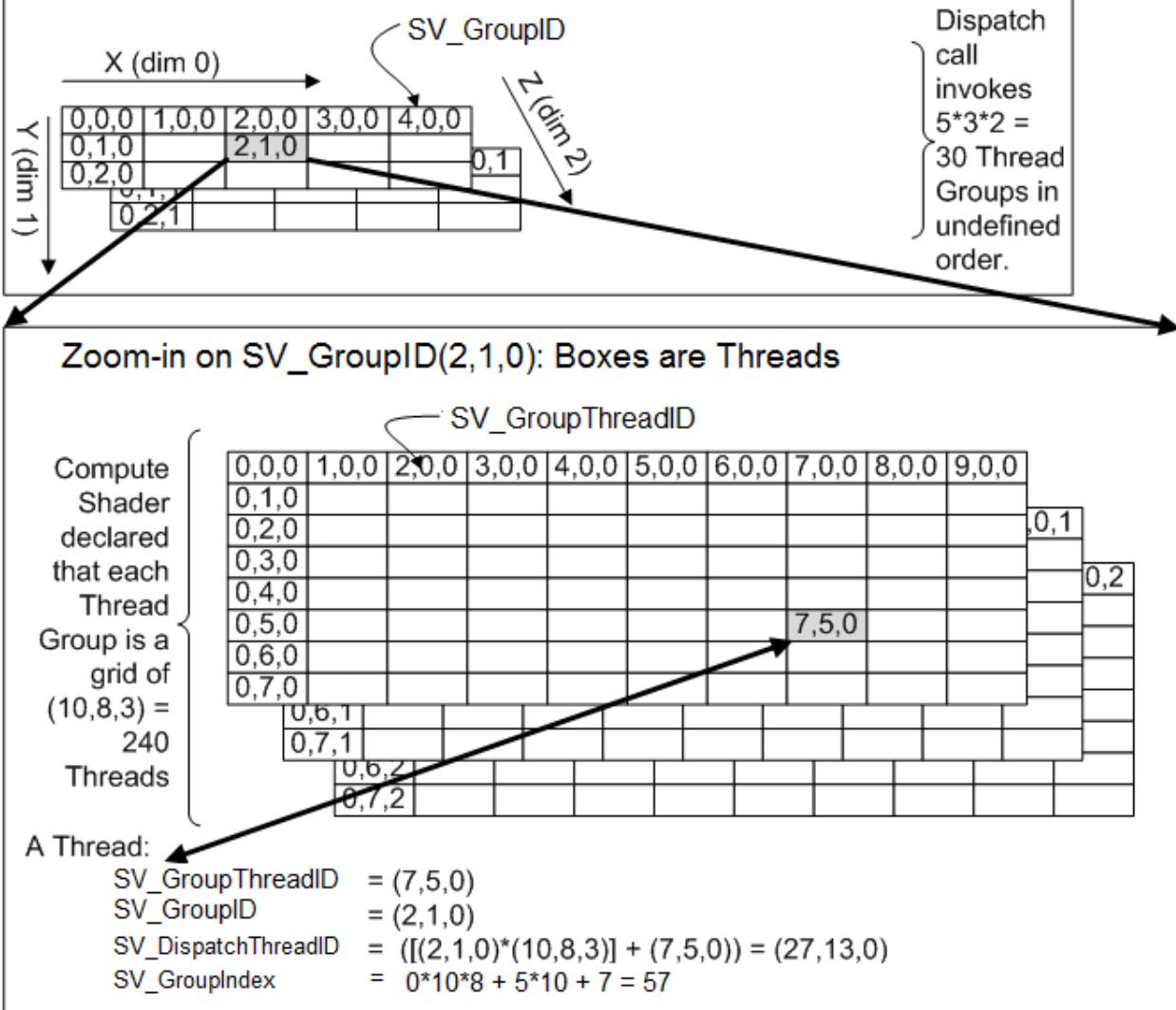
For example, if a compute shader is doing 4x4 matrix addition then numthreads could be set to numthreads(4,4,1) and the indexing in the individual threads would automatically match the matrix entries. The compute shader could also declare a thread group with the same number of threads (16) using numthreads(16,1,1), however it would then have to calculate the current matrix entry based on the current thread number.

The allowable parameter values for **numthreads** depends on the compute shader version.

Compute Shader	Maximum Z	Maximum Threads (X*Y*Z)
cs_4_x	1	768
cs_5_0	64	1024

The following illustration shows the relationship between the parameters passed to [ID3D11DeviceContext::Dispatch](#), Dispatch(5,3,2), the values specified in the numthreads attribute, numthreads(10,8,3), and values that will be passed to the compute shader for the thread related system values ([SV_GroupIndex](#),[SV_DispatchThreadID](#),[SV_GroupThreadID](#),[SV_GroupID](#)).

Dispatch(5,3,2) : Each box below is a Thread Group



This attribute is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	

Related topics

[Shader Model 5 Attributes](#)

[Shader Model 5](#)

outputcontrolpoints

Article • 11/20/2019 • 2 minutes to read

Defines the number of output control points (per thread) that will be created in the hull shader.

```
outputcontrolpoints(X)
```

Remarks

This will be the number of times the main function will be executed.

This attribute is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
	x				

Related topics

[Shader Model 5 Attributes](#)

[Shader Model 5](#)

outputtopology

Article • 11/20/2019 • 2 minutes to read

Defines the output primitive type for the tessellator.

```
outputtopology(X)
```

Remarks

X must be either [point](#), [line](#), [triangle_cw](#), or [triangle_ccw](#) and must be inside quotes. Here are the valid options for this attribute:

```
[outputtopology("point")]
[outputtopology("line")]
[outputtopology("triangle_cw")]
[outputtopology("triangle_ccw")]
```

This attribute is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
			X		

Related topics

[Shader Model 5 Attributes](#)

[Shader Model 5](#)

partitioning

Article • 11/20/2019 • 2 minutes to read

Defines the tessellation scheme to be used in the hull shader.

```
partitioning(X)
```

Remarks

Can be `integer`, `fractional_even`, `fractional_odd`, or `pow2`.

This attribute is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
	x				

Related topics

[Shader Model 5 Attributes](#)

[Shader Model 5](#)

patchconstantfunc

Article • 11/20/2019 • 2 minutes to read

Defines the function for computing patch constant data.

```
patchconstantfunc("function_name")
```

Remarks

function_name is the name of a separate function that outputs the patch-constant data.

This attribute is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
	x				

Related topics

[Shader Model 5 Attributes](#)

[Shader Model 5](#)

Shader Model 5 Intrinsic Functions

Article • 06/08/2022 • 2 minutes to read

Shader Model 5 implements the intrinsic functions from Shader Model 4 and below (see [Intrinsic Functions \(DirectX HLSL\)](#) for a complete list of supported functions), as well as the following new functions:

- [AllMemoryBarrier](#)
- [AllMemoryBarrierWithGroupSync](#)
- [asdouble](#)
- [asuint](#)
- [countbits](#)
- [ddx_coarse](#)
- [ddy_coarse](#)
- [DeviceMemoryBarrier](#)
- [DeviceMemoryBarrierWithGroupSync](#)
- [EvaluateAttributeCentroid](#)
- [EvaluateAttributeAtSample](#)
- [f16tof32](#)
- [f32tof16](#)
- [firstbithigh](#)
- [firstbitlow](#)
- [fma](#)
- [GroupMemoryBarrier](#)
- [GroupMemoryBarrierWithGroupSync](#)
- [InterlockedAdd](#)
- [InterlockedAnd](#)
- [InterlockedCompareExchange](#)
- [InterlockedCompareStore](#)
- [InterlockedExchange](#)
- [InterlockedMax](#)
- [InterlockedMin](#)
- [InterlockedOr](#)
- [InterlockedXor](#)
- [msad4](#)
- [Process2DQuadTessFactorsAvg](#)
- [Process2DQuadTessFactorsMax](#)
- [Process2DQuadTessFactorsMin](#)
- [ProcessIsolineTessFactors](#)
- [ProcessQuadTessFactorsAvg](#)

- [ProcessQuadTessFactorsMax](#)
- [ProcessQuadTessFactorsMin](#)
- [ProcessTriTessFactorsAvg](#)
- [ProcessTriTessFactorsMax](#)
- [ProcessTriTessFactorsMin](#)
- [rcp](#)
- [reversebits](#)

Related topics

[Shader Model 5](#)

Shader Model 5 Objects

Article • 11/20/2019 • 2 minutes to read

Shader Model 5 implements the intrinsic functions from Shader Model 4 and below (see the DirectX SDK), and these new functions, too:

- [AppendStructuredBuffer](#)
- [Buffer](#)
- [ByteAddressBuffer](#)
- [ConsumeStructuredBuffer](#)
- [InputPatch](#)
- [OutputPatch](#)
- [RWBuffer](#)
- [RWByteAddressBuffer](#)
- [RWStructuredBuffer](#)
- [RWTexture1D](#)
- [RWTexture1DArray](#)
- [RWTexture2D](#)
- [RWTexture2DArray](#)
- [RWTexture3D](#)
- [StructuredBuffer](#)
- [Texture1D](#)
- [Texture1DArray](#)
- [Texture2D](#)
- [Texture2DArray](#)
- [Texture2DMS](#)
- [Texture2DMSArray](#)
- [Texture3D](#)
- [TextureCube](#)
- [TextureCubeArray](#)

Related topics

[Shader Model 5](#)

AppendStructuredBuffer

Article • 08/19/2020 • 2 minutes to read

Output buffer that appears as a stream the shader may append to. Only structured buffers can take T types that are structures.

Method	Description
Append	Appends a value to the end of the buffer.
GetDimensions	Gets the resource dimensions.

The UAV format bound to this resource needs to be created with the DXGI_FORMAT_UNKNOWN format.

The UAV bound to this resource must have been created with [D3D11_BUFFER_UAV_FLAG_APPEND](#).

For more information about an append structured buffer, see both sections: [append and consume buffer](#) and [structured buffer](#).

Minimum Shader Model

This object is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This object is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	x

See also

Shader Model 5 Objects

Append function

Article • 11/06/2019 • 2 minutes to read

Appends a value to the end of the buffer.

Syntax

syntax

```
void Append(  
    in T value  
) ;
```

Parameters

value [in]

Type: T

The input value.

Return value

None

Remarks

T can be any data type including structures.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[AppendStructuredBuffer](#)

Shader Model 5

AppendStructuredBuffer::GetDimensions function

Article • 03/09/2021 • 2 minutes to read

Gets the resource dimensions.

Syntax

syntax

```
void GetDimensions(  
    out uint numStructs,  
    out uint stride  
)
```

Parameters

numStructs [out]

Type: **uint**

The number of structures.

stride [out]

Type: **uint**

The number of bytes in each element.

Return value

This function does not return a value.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[AppendStructuredBuffer](#)

[Shader Model 5](#)

Buffer

Article • 10/24/2019 • 2 minutes to read

Buffer type as it exists in Shader Model 4 plus resource variables and buffer info.

Method	Description
GetDimensions	Gets the resource dimensions.
Load	Reads buffer data.
Operator[]	Gets a read-only resource variable.

Minimum Shader Model

This object is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This object is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Shader Model 5 Objects](#)

Buffer::GetDimensions function

Article • 03/28/2023 • 2 minutes to read

The length, in elements, of the Buffer as set in the Shader Resource View.

Syntax

syntax

```
void GetDimensions(  
    out uint dim  
) ;
```

Parameters

dim [out]

Type: **uint**

The length, in bytes, of the buffer.

Return value

This function does not return a value.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Buffer](#)

[Shader Model 5](#)

Buffer::Load methods

Article • 11/06/2019 • 2 minutes to read

Reads [Buffer](#) data.

Overload list

Method	Description
Load(int, uint)	Reads buffer data and returns status of the operation.
Load(int)	Reads buffer data.

See also

[Buffer](#)

Buffer::Load(int, uint) function

Article • 03/09/2021 • 2 minutes to read

Reads buffer data and returns status of the operation.

Syntax

syntax

```
Load(  
    in int Location,  
    out uint Status  
)
```

Parameters

Location [in]

Type: **int**

The location of the buffer.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns **TRUE** if all values from the corresponding [Sample](#), [Gather](#), or [Load](#) operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns **FALSE**.

Return value

Type:

The return type matches the type in the declaration for the [Buffer](#) object.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Examples

This example shows how to use **Load**:

Syntax

```
Buffer<float4> myBuffer;  
float loc;  
uint status;  
float4 myColor = myBuffer.Load( loc , status );
```

See also

[Load methods](#)

Buffer::Operator function

Article • 03/09/2021 • 2 minutes to read

Returns a read-only resource variable.

Syntax

syntax

```
R Operator[](  
    in uint pos  
)
```

Parameters

pos [in]

Type: **uint**

The index position.

Return value

Type: **R**

A read-only resource variable.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	x

See also

[Buffer](#)

Shader Model 5

ByteAddressBuffer

Article • 08/19/2020 • 2 minutes to read

A read-only buffer that is indexed in bytes.

Method	Description
GetDimensions	Gets the resource dimensions.
Load	Gets one value.
Load2	Gets two values.
Load3	Gets three values.
Load4	Gets four values.

You can use the **ByteAddressBuffer** object type when you work with raw buffers. For more info about raw viewing of buffers, see [Raw Views of Buffers](#).

Minimum Shader Model

This object is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models Shader Model 4 (Available through the Direct3D 11 API by using 10.0 or 10.1 feature level (<code>D3D_FEATURE_LEVEL_10_X</code>) on devices that support compute shaders. For more info about compute shader support on downlevel hardware, see Compute Shaders on Downlevel Hardware .)	yes

This object is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

For more info about a byte address buffer, see the [byte addressable resource type](#).

Shader Model 5 also implements a [read-write byte address buffer](#).

See also

[Shader Model 5 Objects](#)

ByteAddressBuffer::GetDimensions function

Article • 03/09/2021 • 2 minutes to read

Gets the length of the buffer.

Syntax

syntax

```
void GetDimensions(  
    out uint dim  
) ;
```

Parameters

dim [out]

Type: `uint`

The length, in bytes, of the buffer.

Return value

This function does not return a value.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[ByteAddressBuffer](#)

Shader Model 5

Load methods

Article • 11/06/2019 • 2 minutes to read

Gets one value from a read-only buffer indexed in bytes.

Overload list

Method	Description
Load(int, uint)	Gets one value and the status of the operation.
Load(int)	Gets one value.

See also

[ByteAddressBuffer](#)

ByteAddressBuffer::Load(int) function

Article • 03/09/2021 • 2 minutes to read

Gets one value.

Syntax

syntax

```
uint Load(  
    in int address  
) ;
```

Parameters

address [in]

Type: **int**

The input address in bytes, which must be a multiple of 4.

Return value

Type: **uint**

One value.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Load methods](#)

Shader Model 5

ByteAddressBuffer::Load(int, uint) function

Article • 12/10/2020 • 2 minutes to read

Gets one value and the status of the operation.

Syntax

syntax

```
uint Load(  
    in int Location,  
    out uint Status  
) ;
```

Parameters

Location [in]

Type: **int**

The input address in bytes, which must be a multiple of 4.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. **CheckAccessFullyMapped** returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, **CheckAccessFullyMapped** returns **FALSE**.

Return value

Type: **uint**

One value.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

See also

[Load methods](#)

[**ByteAddressBuffer**](#)

Load2 methods

Article • 11/06/2019 • 2 minutes to read

Gets two values from a read-only buffer indexed in bytes.

Overload list

Method	Description
Load2(uint)	Gets two values.
Load2(uint, uint)	Gets two values and the status of the operation.

See also

[ByteAddressBuffer](#)

ByteAddressBuffer::Load2(uint) function

Article • 03/09/2021 • 2 minutes to read

Gets two values.

Syntax

syntax

```
uint2 Load2(  
    in uint address  
) ;
```

Parameters

address [in]

Type: **uint**

The input address in bytes, which must be a multiple of 4.

Return value

Type: **uint2**

Two values.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Load2 methods](#)

Shader Model 5

Load2(uint, uint) function

Article • 08/19/2020 • 2 minutes to read

Gets two values and the status of the operation.

Syntax

syntax

```
uint2 Load2(  
    in uint Location,  
    out uint Status  
)
```

Parameters

Location [in]

Type: **uint**

The input address in bytes, which must be a multiple of 4.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns **TRUE** if all values from the corresponding [Sample](#), [Gather](#), or [Load](#) operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns **FALSE**.

Return value

Type: **uint2**

Two values.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

See also

[Load2 methods](#)

[ByteAddressBuffer](#)

Load3 methods

Article • 11/06/2019 • 2 minutes to read

Gets three values from a read-only buffer indexed in bytes.

Overload list

Method	Description
Load3(uint)	Gets three values.
Load3(uint, uint)	Gets three values and the status of the operation.

See also

[ByteAddressBuffer](#)

ByteAddressBuffer::Load3(uint) function

Article • 03/09/2021 • 2 minutes to read

Gets three values.

Syntax

syntax

```
uint3 Load3(  
    in uint address  
) ;
```

Parameters

address [in]

Type: **uint**

The input address in bytes, which must be a multiple of 4.

Return value

Type: **uint3**

Three values.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Load3 methods](#)

Shader Model 5

Load3(uint, uint) function

Article • 08/19/2020 • 2 minutes to read

Gets three values and the status of the operation.

Syntax

syntax

```
uint3 Load3(  
    in  uint Location,  
    out uint Status  
) ;
```

Parameters

Location [in]

Type: **uint**

The input address in bytes, which must be a multiple of 4.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. **CheckAccessFullyMapped** returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, **CheckAccessFullyMapped** returns **FALSE**.

Return value

Type: **uint3**

Three values.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

See also

[Load3 methods](#)

[ByteAddressBuffer](#)

Load4 methods

Article • 11/06/2019 • 2 minutes to read

Gets four values from a read-only buffer indexed in bytes.

Overload list

Method	Description
Load4(uint)	Gets four values.
Load4(uint, uint)	Gets four values and the status of the operation.

See also

[ByteAddressBuffer](#)

ByteAddressBuffer::Load4(uint) function

Article • 03/09/2021 • 2 minutes to read

Gets four values.

Syntax

syntax

```
uint4 Load4(  
    in uint address  
) ;
```

Parameters

address [in]

Type: **uint**

The input address in bytes, which must be a multiple of 4.

Return value

Type: **uint4**

Four values.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Load4 methods](#)

Shader Model 5

Load4(uint, uint) function

Article • 08/19/2020 • 2 minutes to read

Gets four values and the status of the operation.

Syntax

syntax

```
uint4 Load4(  
    in uint Location,  
    out uint Status  
);
```

Parameters

Location [in]

Type: **uint**

The input address in bytes, which must be a multiple of 4.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns **TRUE** if all values from the corresponding [Sample](#), [Gather](#), or [Load](#) operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns **FALSE**.

Return value

Type: **uint4**

Four values.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

See also

[Load4 methods](#)

[ByteAddressBuffer](#)

ConsumeStructuredBuffer

Article • 08/19/2020 • 2 minutes to read

An input buffer that appears as a stream the shader may pull values from. Only structured buffers can take T types that are structures.

Method	Description
Consume	Removes a value from the end of the buffer.
GetDimensions	Gets the resource dimensions.

The UAV format bound to this resource needs to be created with the DXGI_FORMAT_UNKNOWN format.

The UAV bound to this resource must have been created with [D3D11_BUFFER_UAV_FLAG_APPEND](#).

For more info about consume structured buffers, see both sections: [append and consume buffer](#) and [structured buffer](#).

Minimum Shader Model

This object is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This object is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	x

See also

Shader Model 5 Objects

Consume function

Article • 11/06/2019 • 2 minutes to read

Removes a value from the end of the buffer.

Syntax

syntax

```
T Consume(void);
```

Parameters

This function has no parameters.

Return value

Type: T

The value removed (can be a structure).

Remarks

T can be any data type including a structure.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[ConsumeStructuredBuffer](#)

[Shader Model 5](#)

ConsumeStructuredBuffer::GetDimensions function

Article • 03/09/2021 • 2 minutes to read

Gets the resource dimensions.

Syntax

syntax

```
void GetDimensions(  
    out uint numStructs,  
    out uint stride  
) ;
```

Parameters

numStructs [out]

Type: **uint**

The number of structures.

stride [out]

Type: **uint**

The stride, in bytes, of each element.

Return value

This function does not return a value.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[ConsumeStructuredBuffer](#)

[Shader Model 5](#)

InputPatch

Article • 10/24/2019 • 2 minutes to read

Represents an array of control points that are available to the hull shader as inputs.

Method	Description
Operator[]	Gets a read-only resource variable.

The InputPatch class also supports the following properties:

Properties	Type	Description
Length	uint	The number of control points.

Minimum Shader Model

This object is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This object is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
	x		x		

See also

[Shader Model 5 Objects](#)

InputPatch::Operator function

Article • 03/09/2021 • 2 minutes to read

Returns the n^{th} control point in the patch.

Syntax

syntax

```
T Operator[](  
    in uint n  
);
```

Parameters

n [in]

Type: **uint**

The input index.

Return value

Type: **T**

The n^{th} control point.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
	x				

See also

[InputPatch](#)

Shader Model 5

OutputPatch

Article • 10/24/2019 • 2 minutes to read

Represents an array of output control points that are available to the hull shader's patch-constant function as well as the domain shader.

Method	Description
Operator[]	Gets a read-only resource variable.

In addition, the InputPatch class supports the following properties:

Properties	Type	Description
Length	uint	The number of control points.

Minimum Shader Model

This object is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This object is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
	x	x			

See also

[Shader Model 5 Objects](#)

OutputPatch::Operator function

Article • 03/09/2021 • 2 minutes to read

Returns the n^{th} control point in the patch.

Syntax

syntax

```
T Operator[](  
    in uint n  
) ;
```

Parameters

n [in]

Type: `uint`

The input index.

Return value

Type: `T`

The n^{th} control point.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
	x	x			

See also

[OutputPatch](#)

Shader Model 5

RWBuffer

Article • 10/24/2019 • 2 minutes to read

A read/write buffer.

Method	Description
GetDimensions	Gets the resource dimensions.
Load	Gets one value in a read-write buffer.
Operator[]	Returns a resource variable.

A resource variable can also be passed into any unordered or interlocked operation.

RWBuffer objects can be prefixed with the storage class **globallycoherent**. This storage class causes memory barriers and syncs to flush data across the entire GPU such that other groups can see writes. Without this specifier, a memory barrier or sync will flush a UAV only within the current group.

Minimum Shader Model

This object is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This object is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

See also

[Shader Model 5 Objects](#)

RWBuffer::Load methods

Article • 11/06/2019 • 2 minutes to read

Gets one value from a [RWBuffer](#).

Overload list

Method	Description
Load(int,uint)	Reads buffer data and returns status of the operation.
Load(int)	Reads buffer data.

See also

[RWBuffer](#)

RWBuffer::Load(int) function

Article • 03/09/2021 • 2 minutes to read

Reads buffer data.

Syntax

syntax

```
Load(  
    in int Location  
) ;
```

Parameters

Location [in]

Type: **int**

The location of the buffer.

Return value

Type:

The return type matches the type in the declaration for the **RWBuffer** object.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Load methods](#)

RWBuffer::Load(int,uint) function

Article • 03/09/2021 • 2 minutes to read

Reads buffer data and returns status of the operation.

Syntax

syntax

```
Load(  
    in int Location,  
    out uint Status  
)
```

Parameters

Location [in]

Type: **int**

The location of the buffer.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. **CheckAccessFullyMapped** returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, **CheckAccessFullyMapped** returns **FALSE**.

Return value

Type:

The return type matches the type in the declaration for the [RWBuffer](#) object.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

See also

[Load methods](#)

RWBuffer::GetDimensions function

Article • 03/28/2023 • 2 minutes to read

Gets the length of the buffer.

Syntax

syntax

```
void GetDimensions(  
    out uint dim  
) ;
```

Parameters

dim [out]

Type: **uint**

The length, in elements, of the Buffer as set in the Unordered Resource View.

Return value

Nothing

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	x

See also

[RWBuffer](#)

[Shader Model 5](#)

RWBuffer::Operator function

Article • 03/09/2021 • 2 minutes to read

Returns a resource variable.

Syntax

syntax

```
R Operator[](  
    in uint pos  
)
```

Parameters

pos [in]

Type: **uint**

The index position.

Return value

Type: **R**

A resource variable.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[RWBuffer](#)

Shader Model 5

RWByteAddressBuffer

Article • 08/19/2020 • 2 minutes to read

A read/write buffer that indexes in bytes.

Method	Description
GetDimensions	Gets the resource dimensions.
InterlockedAdd	Adds, atomically.
InterlockedAnd	ANDs, atomically.
InterlockedCompareExchange	Compares and exchanges, atomically.
InterlockedCompareStore	Compares and stores, atomically.
InterlockedExchange	Exchanges, atomically.
InterlockedMax	Finds the max, atomically.
InterlockedMin	Find the min, atomically.
InterlockedOr	ORs, atomically.
InterlockedXor	XORs, atomically.
Load	Gets one value.
Load2	Gets two values.
Load3	Gets three values.
Load4	Gets four values.
Store	Sets one value.
Store2	Sets two values.
Store3	Sets three values.
Store4	Sets four values.

RWByteAddressBuffer objects can be prefixed with the storage class **globallycoherent**. This storage class causes memory barriers and syncs to flush data across the entire GPU such that other groups can see writes. Without this specifier, a memory barrier or sync will flush a UAV only within the current group.

The UAV format bound to this resource needs to be created with the DXGI_FORMAT_R32_TYPELESS format.

The UAV bound to this resource must have been created with the [D3D11_BUFFER_UAV_FLAG_RAW](#).

You can use the [RWByteAddressBuffer](#) object type when you work with raw buffers. For more info about raw viewing of buffers, see [Raw Views of Buffers](#).

Minimum Shader Model

This object is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models Shader Model 4 (Available through the Direct3D 11 API by using 10.0 or 10.1 feature level (D3D_FEATURE_LEVEL_10_X) on devices that support compute shaders. For more information about compute shader support on downlevel hardware, see Compute Shaders on Downlevel Hardware .)	yes

This object is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

See also

[Shader Model 5 Objects](#)

RWByteAddressBuffer::GetDimensions function

Article • 03/09/2021 • 2 minutes to read

Gets the length of the buffer.

Syntax

syntax

```
void GetDimensions(  
    out uint dim  
) ;
```

Parameters

dim [out]

Type: `uint`

The length, in bytes, of the buffer.

Return value

This function does not return a value.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	x

See also

[RWByteAddressBuffer](#)

Shader Model 5

InterlockedAdd function

Article • 08/19/2020 • 2 minutes to read

Adds the value, atomically.

Syntax

syntax

```
void InterlockedAdd(
    in  UINT dest,
    in  UINT value,
    out UINT original_value
);
```

Parameters

dest [in]

Type: [UINT](#)

The destination address.

value [in]

Type: [UINT](#)

The input value.

original_value [out]

Type: [UINT](#)

The original value.

Return value

This function does not return a value.

Remarks

This operation can be performed only on int or uint typed resources and shared memory variables. There are three possible uses for this function. The first is when R is a shared memory variable type. In this case, the function performs an atomic add of value to the shared memory register referenced by dest. The second scenario is when R is a resource variable type. In this scenario, the function performs an atomic add of the value to the resource location referenced by dest. Finally, the third scenario is when R is a local variable type. In this scenario, the function reduces to a sum of the value of dest and value, stored in dest. The overloaded function has an additional output variable which will be set to the original value of dest. This overloaded operation is only available when R is readable and writable.

This function is supported in the following types of shaders:

VS	HS	DS	GS	PS	CS
x	x	x	x	x	x

Requirements

See also

[RWByteAddressBuffer](#)

[Shader Model 5](#)

InterlockedAnd function

Article • 08/19/2020 • 2 minutes to read

Ands the value, atomically.

Syntax

syntax

```
void InterlockedAnd(
    in  UINT dest,
    in  UINT value,
    out UINT original_value
);
```

Parameters

dest [in]

Type: [UINT](#)

The destination address.

value [in]

Type: [UINT](#)

The input value.

original_value [out]

Type: [UINT](#)

The original value.

Return value

This function does not return a value.

Remarks

This operation can only be performed on int or uint typed resources and shared memory variables. There are three possible uses for this function. The first is when R is a shared memory variable type. In this case, the function performs an atomic and of value to the shared memory register referenced by dest. The second scenario is when R is a resource variable type. In this scenario, the function performs an atomic and of value to the resource location referenced by dest. Finally, the third scenario is when R is a local variable type. In this scenario, the function reduces to an and of the value of dest and value, stored in dest. The overloaded function has an additional output variable which will be set to the original value of dest. This overloaded operation is only available when R is readable and writable.

This function is supported in the following types of shaders:

VS	HS	DS	GS	PS	CS
x	x	x	x	x	x

See also

[RWByteAddressBuffer](#)

[Shader Model 5](#)

InterlockedCompareExchange function

Article • 08/19/2020 • 2 minutes to read

Compares the input to the comparison value and exchanges the result, atomically.

Syntax

```
syntax

void InterlockedCompareExchange(
    in  UINT dest,
    in  UINT compare_value,
    in  UINT value,
    out UINT original_value
);
```

Parameters

dest [in]

Type: [UINT](#)

The destination address.

compare_value [in]

Type: [UINT](#)

The comparison value.

value [in]

Type: [UINT](#)

The input value.

original_value [out]

Type: [UINT](#)

The original value.

Return value

This function does not return a value.

Remarks

Atomically compares the value in *dest* to *compare_value*, stores value in *dest* if the values match, returns the original value of *dest* in *original_value*. This operation can only be performed on **int** or **uint** typed resources and shared memory variables. There are three possible uses for this function. The first is when R is a shared memory variable type. In this case, the function performs the operation on the shared memory register referenced by *dest*. The second scenario is when R is a resource variable type. In this scenario, the function performs the operation on the resource location referenced by *dest*. Finally, the third scenario is when R is a local variable type. In this scenario, the function reduces to the operation performed using local operations. This operation is only available when R is readable and writable.

ⓘ Note

If you call **InterlockedCompareExchange** in a **for** or **while** compute shader loop, to properly compile, you must use the **[allow_uav_condition]** attribute on that loop.

This function is supported in the following types of shaders:

VS	HS	DS	GS	PS	CS
x	x	x	x	x	x

See also

[RWByteAddressBuffer](#)

[Shader Model 5](#)

InterlockedCompareStore function

Article • 04/20/2022 • 2 minutes to read

Compares the input to the comparison value, atomically.

Syntax

syntax

```
void InterlockedCompareStore(
    in UINT dest,
    in UINT compare_value,
    in UINT value
);
```

Parameters

dest [in]

Type: [UINT](#)

The destination address.

compare_value [in]

Type: [UINT](#)

The comparison value.

value [in]

Type: [UINT](#)

The input value.

Return value

This function does not return a value.

Remarks

This operation can only be performed on int or uint typed resources and shared memory variables. There are three possible uses for this function. The first is when R is a shared memory variable type. In this case, the function performs the operation on the shared memory register referenced by dest. The second scenario is when R is a resource variable type. In this scenario, the function performs the operation on the resource location referenced by dest. Finally, the third scenario is when R is a local variable type. In this scenario, the function reduces to the operation performed using local operations.

This function is supported in the following types of shaders:

VS	HS	DS	GS	PS	CS
x	x	x	x	x	x

See also

[RWByteAddressBuffer](#)

[Shader Model 5](#)

InterlockedExchange function

Article • 08/19/2020 • 2 minutes to read

Exchanges a value, atomically.

Syntax

syntax

```
void InterlockedExchange(
    in UINT dest,
    in UINT value,
    out UINT original_value
);
```

Parameters

dest [in]

Type: [UINT](#)

The destination address.

value [in]

Type: [UINT](#)

The input value.

original_value [out]

Type: [UINT](#)

The original value.

Return value

This function does not return a value.

Remarks

This operation can only be performed on scalar-typed resources and shared memory variables. There are three possible uses for this function. The first is when R is a shared memory variable type. In this case, the function performs the operation on the shared memory register referenced by dest. The second scenario is when R is a resource variable type. In this scenario, the function performs the operation on the resource location referenced by dest. Finally, the third scenario is when R is a local variable type. In this scenario, the function reduces to the operation performed using local operations. This operation is only available when R is readable and writable.

This function is supported in the following types of shaders:

VS	HS	DS	GS	PS	CS
x	x	x	x	x	x

See also

[RWByteAddressBuffer](#)

[Shader Model 5](#)

InterlockedMax function

Article • 08/19/2020 • 2 minutes to read

Finds the maximum value, atomically.

Syntax

syntax

```
void InterlockedMax(
    in  UINT dest,
    in  UINT value,
    out UINT original_value
);
```

Parameters

dest [in]

Type: [UINT](#)

The destination address.

value [in]

Type: [UINT](#)

The input value.

original_value [out]

Type: [UINT](#)

The original value.

Return value

This function does not return a value.

Remarks

This operation can only be performed on int and uint typed resources and shared memory variables. There are three possible uses for this function. The first is when R is a shared memory variable type. In this case, the function performs an atomic maximum of the value to the shared memory register referenced by dest. The second scenario is when R is a resource variable type. In this scenario, the function performs an atomic maximum of the value to the resource location referenced by dest. Finally, the third scenario is when R is a local variable type. In this scenario, the function reduces to a maximum of the value of dest and value, stored in dest. The overloaded function has an additional output variable which will be set to the original value of dest. This overloaded operation is only available when R is readable and writable.

This function is supported in the following types of shaders:

VS	HS	DS	GS	PS	CS
x	x	x	x	x	x

See also

[RWByteAddressBuffer](#)

[Shader Model 5](#)

InterlockedMin function

Article • 08/19/2020 • 2 minutes to read

Finds the minimum value, atomically.

Syntax

syntax

```
void InterlockedMin(
    in  UINT dest,
    in  UINT value,
    out UINT original_value
);
```

Parameters

dest [in]

Type: [UINT](#)

The destination address.

value [in]

Type: [UINT](#)

The input value.

original_value [out]

Type: [UINT](#)

The original value.

Return value

Nothing

Remarks

This operation can only be performed on int and uint typed resources and shared memory variables. There are three possible uses for this function. The first is when R is a shared memory variable type. In this case, the function performs an atomic minimum of the value to the shared memory register referenced by dest. The second scenario is when R is a resource variable type. In this scenario, the function performs an atomic minimum of the value to the resource location referenced by dest. Finally, the third scenario is when R is a local variable type. In this scenario, the function reduces to a minimum of the value of dest and value, stored in dest. The overloaded function has an additional output variable which will be set to the original value of dest. This overloaded operation is only available when R is readable and writable.

This function is supported in the following types of shaders:

VS	HS	DS	GS	PS	CS
x	x	x	x	x	x

See also

[RWByteAddressBuffer](#)

[Shader Model 5](#)

InterlockedOr function

Article • 08/19/2020 • 2 minutes to read

Performs an atomic **OR** on the value.

Syntax

syntax

```
void InterlockedOr(
    in  UINT dest,
    in  UINT value,
    out UINT original_value
);
```

Parameters

dest [in]

Type: [UINT](#)

The destination address.

value [in]

Type: [UINT](#)

The input value.

original_value [out]

Type: [UINT](#)

The original value.

Return value

Nothing

Remarks

This operation can be performed only on **INT** or **UINT** typed resources and shared memory variables. There are three possible uses for this function. The first is when R is a shared memory variable type. In this case, the function performs an atomic **OR** with the value of the shared memory register referenced by *dest*. The second scenario is when R is a resource variable type. In this scenario, the function performs an atomic **OR** with the value of the resource location referenced by *dest*. Finally, the third scenario is when R is a local variable type. In this scenario, the function reduces to an **OR** with the values of *dest* and *value*. The result of the operation replaces the value of *dest*. The overloaded function has an additional output variable, which will be set to the original value of *dest*. This overloaded operation is available only when R is readable and writable.

This function is supported in the following types of shaders:

VS	HS	DS	GS	PS	CS
x	x	x	x	x	x

See also

[RWByteAddressBuffer](#)

[Shader Model 5](#)

InterlockedXor function

Article • 08/19/2020 • 2 minutes to read

Performs an atomic **XOR** on the value.

Syntax

syntax

```
void InterlockedXor(
    in  UINT dest,
    in  UINT value,
    out UINT original_value
);
```

Parameters

dest [in]

Type: [UINT](#)

The destination address.

value [in]

Type: [UINT](#)

The input value.

original_value [out]

Type: [UINT](#)

The original value.

Return value

This function does not return a value.

Remarks

This operation can be performed only on **INT** or **UINT** typed resources and shared memory variables. There are three possible uses for this function. The first is when R is a shared memory variable type. In this case, the function performs an atomic **XOR** with the value of the shared memory register referenced by *dest*. The second scenario is when R is a resource variable type. In this scenario, the function performs an atomic **XOR** with the value of the resource location referenced by *dest*. Finally, the third scenario is when R is a local variable type. In this scenario, the function reduces to an **XOR** of the values of *dest* and *value*. The result of the operation replaces the value in *dest*. The overloaded function has an additional output variable which will be set to the original value of *dest*. This overloaded operation is available only when R is readable and writable.

This function is supported in the following types of shaders:

VS	HS	DS	GS	PS	CS
x	x	x	x	x	x

See also

[RWByteAddressBuffer](#)

[Shader Model 5](#)

RWByteAddressBuffer::Load methods

Article • 11/06/2019 • 2 minutes to read

Gets one value from a [RWByteAddressBuffer](#).

Overload list

Method	Description
Load(int,uint)	Gets one value and returns status of the operation.
Load(int)	Gets one value.

See also

[RWByteAddressBuffer](#)

RWByteAddressBuffer::Load(int) function

Article • 03/09/2021 • 2 minutes to read

Gets one value.

Syntax

syntax

```
uint Load(  
    in int Location  
>;
```

Parameters

Location [in]

Type: **int**

The location of the buffer.

Return value

Type: **uint**

One value.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	x

See also

Load methods

RWByteAddressBuffer::Load(int,uint) function

Article • 12/10/2020 • 2 minutes to read

Gets one value and returns status of the operation.

Syntax

syntax

```
uint Load(  
    in int Location,  
    out uint Status  
) ;
```

Parameters

Location [in]

Type: **int**

The location of the buffer.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. **CheckAccessFullyMapped** returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, **CheckAccessFullyMapped** returns **FALSE**.

Return value

Type: **uint**

One value.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

See also

[Load methods](#)

RWByteAddressBuffer::Load2 methods

Article • 11/06/2019 • 2 minutes to read

Gets two values from a [RWByteAddressBuffer](#).

Overload list

Method	Description
Load2(uint)	Gets two values.
Load2(uint,uint)	Gets two values and returns status of the operation.

See also

[RWByteAddressBuffer](#)

RWByteAddressBuffer::Load2(uint) function

Article • 03/09/2021 • 2 minutes to read

Gets two values.

Syntax

syntax

```
uint2 Load2(  
    in uint address  
) ;
```

Parameters

address [in]

Type: **uint**

The input address in bytes, which must be a multiple of 4.

Return value

Type: **uint2**

Two values.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	x

See also

Load2 methods

Shader Model 5

Load2(uint,uint) function

Article • 08/19/2020 • 2 minutes to read

Gets two values and returns status of the operation.

Syntax

syntax

```
uint2 Load2(  
    in uint Location,  
    out uint Status  
);
```

Parameters

Location [in]

Type: **uint**

The location of the buffer.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. **CheckAccessFullyMapped** returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, **CheckAccessFullyMapped** returns **FALSE**.

Return value

Type: **uint2**

Two values.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

See also

[Load2 methods](#)

RWByteAddressBuffer::Load3 methods

Article • 11/06/2019 • 2 minutes to read

Gets three values from a [RWByteAddressBuffer](#).

Overload list

Method	Description
Load3(uint)	Gets three values.
Load3(uint,uint)	Gets three values and returns status of the operation.

See also

[RWByteAddressBuffer](#)

RWByteAddressBuffer::Load3(uint) function

Article • 03/09/2021 • 2 minutes to read

Gets three values.

Syntax

syntax

```
uint3 Load3(  
    in uint address  
) ;
```

Parameters

address [in]

Type: **uint**

The input address in bytes, which must be a multiple of 4.

Return value

Type: **uint3**

Three values.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	x

See also

Load3 methods

Shader Model 5

Load3(uint,uint) function

Article • 08/19/2020 • 2 minutes to read

Gets three values and returns status of the operation.

Syntax

syntax

```
uint3 Load3(  
    in uint Location,  
    out uint Status  
);
```

Parameters

Location [in]

Type: **uint**

The location of the buffer.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. **CheckAccessFullyMapped** returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, **CheckAccessFullyMapped** returns **FALSE**.

Return value

Type: **uint3**

Three values.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

See also

[Load3 methods](#)

RWByteAddressBuffer::Load4 methods

Article • 11/06/2019 • 2 minutes to read

Gets four values from a [RWByteAddressBuffer](#).

Overload list

Method	Description
Load4(uint)	Gets four values.
Load4(uint,uint)	Gets four values and returns status of the operation.

See also

[RWByteAddressBuffer](#)

RWByteAddressBuffer::Load4(uint) function

Article • 03/09/2021 • 2 minutes to read

Gets four values.

Syntax

syntax

```
uint4 Load4(  
    in uint address  
) ;
```

Parameters

address [in]

Type: **uint**

The input address in bytes, which must be a multiple of 4.

Return value

Type: **uint4**

Four values.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	x

See also

Load4 methods

Shader Model 5

Load4(uint,uint) function

Article • 08/19/2020 • 2 minutes to read

Gets four values and returns status of the operation.

Syntax

syntax

```
uint4 Load4(  
    in uint Location,  
    out uint Status  
);
```

Parameters

Location [in]

Type: **uint**

The location of the buffer.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. **CheckAccessFullyMapped** returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, **CheckAccessFullyMapped** returns **FALSE**.

Return value

Type: **uint4**

Four values.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

See also

[Load4 methods](#)

Store function

Article • 11/06/2019 • 2 minutes to read

Sets one value.

Syntax

syntax

```
void Store(  
    in uint address,  
    in uint value  
)
```

Parameters

address [in]

Type: **uint**

The input address in bytes, which must be a multiple of 4.

value [in]

Type: **uint**

One input value.

Return value

This function does not return a value.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

See also

[RWByteAddressBuffer](#)

[Shader Model 5](#)

Store2 function

Article • 11/06/2019 • 2 minutes to read

Sets two values.

Syntax

syntax

```
void Store2(  
    in uint address,  
    in uint2 values  
);
```

Parameters

address [in]

Type: **uint**

The input address in bytes, which must be a multiple of 4.

values [in]

Type: **uint2**

Two input values.

Return value

This function does not return a value.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

See also

[RWByteAddressBuffer](#)

[Shader Model 5](#)

Store3 function

Article • 11/06/2019 • 2 minutes to read

Sets three values.

Syntax

syntax

```
void Store3(  
    in uint address,  
    in uint3 values  
);
```

Parameters

address [in]

Type: **uint**

The input address in bytes, which must be a multiple of 4.

values [in]

Type: **uint3**

Three input values.

Return value

This function does not return a value.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

See also

[RWByteAddressBuffer](#)

[Shader Model 5](#)

Store4 function

Article • 11/06/2019 • 2 minutes to read

Sets four values.

Syntax

syntax

```
void Store4(  
    in uint address,  
    in uint4 values  
);
```

Parameters

address [in]

Type: **uint**

The input address in bytes, which must be a multiple of 4.

values [in]

Type: **uint4**

Four input values.

Return value

This function does not return a value.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

See also

[RWByteAddressBuffer](#)

[Shader Model 5](#)

RWStructuredBuffer

Article • 08/19/2020 • 2 minutes to read

A read/write buffer that can take a T type that is a structure.

Method	Description
DecrementCounter	Decrements the object's hidden counter.
GetDimensions	Gets the resource dimensions.
IncrementCounter	Increments the object's hidden counter.
Load	Reads buffer data.
Operator[]	Returns a resource variable.

A resource variable can also be passed into any unordered or interlocked operation.

RWStructuredBuffer objects can be prefixed with the storage class **globallycoherent**. This storage class causes memory barriers and syncs to flush data across the entire GPU such that other groups can see writes. Without this specifier, a memory barrier or sync will only flush a UAV within the current group.

The UAV format bound to this resource needs to be created with the DXGI_FORMAT_UNKNOWN format.

To find out more about [structured buffers](#), see the overview material.

Minimum Shader Model

This object is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models Shader Model 4 (Available through the Direct3D 11 API by using 10.0 or 10.1 feature level (D3D_FEATURE_LEVEL_10_X) on devices that support compute shaders. For more information about compute shader support on downlevel hardware, see Compute Shaders on Downlevel Hardware .)	yes

This object is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

See also

[Shader Model 5 Objects](#)

DecrementCounter function

Article • 08/15/2022 • 2 minutes to read

Decrements the object's hidden counter.

Syntax

syntax

```
uint DecrementCounter(void);
```

Parameters

This function has no parameters.

Return value

Type: **uint**

The post-decremented counter value.

Remarks

The bound unordered access view must have [D3D11_BUFFER_UAV_FLAG_COUNTER](#) set during creation for this method to work.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	x

See also

[RWStructuredBuffer](#)

[Shader Model 5](#)

RWStructuredBuffer::GetDimensions function

Article • 03/09/2021 • 2 minutes to read

Gets the resource dimensions.

Syntax

syntax

```
void GetDimensions(  
    out uint numStructs,  
    out uint stride  
)
```

Parameters

numStructs [out]

Type: **uint**

The number of structures in the resource.

stride [out]

Type: **uint**

The stride, in bytes, of each structure element.

Return value

Nothing

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

See also

[RWStructuredBuffer](#)

[Shader Model 5](#)

IncrementCounter function

Article • 08/15/2022 • 2 minutes to read

Increments the object's hidden counter.

Syntax

syntax

```
uint IncrementCounter(void);
```

Parameters

This function has no parameters.

Return value

Type: **uint**

The pre-incremented counter value.

Remarks

The bound unordered access view must have [D3D11_BUFFER_UAV_FLAG_COUNTER](#) set during creation for this method to work.

A structured resource can be further indexed based on the component names of the structures.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	x

See also

[RWStructuredBuffer](#)

Shader Model 5

RWStructuredBuffer::Load methods

Article • 11/06/2019 • 2 minutes to read

Reads buffer data into a [RWStructuredBuffer](#).

Overload list

Method	Description
Load(int,uint)	Reads buffer data and returns status about the operation.
Load(int)	Reads buffer data.

See also

[RWStructuredBuffer](#)

RWStructuredBuffer::Load(int) function

Article • 03/09/2021 • 2 minutes to read

Reads buffer data.

Syntax

syntax

```
Load(  
    in int Location  
) ;
```

Parameters

Location [in]

Type: **int**

The location of the buffer.

Return value

Type:

The return type matches the type in the declaration for the [RWStructuredBuffer](#) object.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Load methods](#)

RWStructuredBuffer::Load(int,uint) function

Article • 03/09/2021 • 2 minutes to read

Reads buffer data and returns status about the operation.

Syntax

syntax

```
Load(  
    in int Location,  
    out uint Status  
)
```

Parameters

Location [in]

Type: **int**

The location of the buffer.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns **TRUE** if all values from the corresponding [Sample](#), [Gather](#), or [Load](#) operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns **FALSE**.

Return value

Type:

The return type matches the type in the declaration for the [RWStructuredBuffer](#) object.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

See also

[Load methods](#)

RWStructuredBuffer::Operator function

Article • 03/09/2021 • 2 minutes to read

Returns a resource variable.

Syntax

syntax

```
R Operator[](  
    in uint pos  
)
```

Parameters

pos [in]

Type: **uint**

The index position.

Return value

Type: **R**

A resource variable.

Remarks

A structured resource can be further indexed based on the component names of the structures.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	x

See also

[RWStructuredBuffer](#)

[Shader Model 5](#)

RWTexture1D

Article • 03/09/2021 • 2 minutes to read

A read/write resource.

Method	Description
GetDimensions	Gets the resource dimensions.
Load	Reads texture data.
Operator[]	Gets a resource variable.

You can prefix **RWTexture1D** objects with the storage class **globallycoherent**. This storage class causes memory barriers and syncs to flush data across the entire GPU such that other groups can see writes. Without this specifier, a memory barrier or sync will flush a UAV only within the current group.

A **RWTexture1D** object requires an element type in a declaration statement for the object. For example, the following declaration is correct:

```
RWTexture1D<float> tex;
```

Because a **RWTexture1D** object is a UAV-type object, its properties differ from a shader resource view (SRV)-type object, such as a **Texture1D** object. For example, you can read from and write to a **RWTexture1D** object, but you can only read from a **Texture1D** object.

A **RWTexture1D** object cannot use methods from a **Texture1D** object, such as [Sample](#). However, because you can create multiple view types to the same resource, you can declare multiple texture types as a single texture in multiple shaders. For example, you can declare and use a **RWTexture1D** object as *tex* in a compute shader and then declare and use a **Texture1D** object as *tex* in a pixel shader.

ⓘ Note

The runtime enforces certain usage patterns when you create multiple view types to the same resource. For example, the runtime does not allow you to have both a

UAV mapping for a resource and SRV mapping for the same resource active at the same time.

Minimum Shader Model

This object is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This object is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	x

See also

[Shader Model 5 Objects](#)

RWTexture1D::GetDimensions function

Article • 03/09/2021 • 2 minutes to read

Returns the dimensions of the resource.

Syntax

syntax

```
void GetDimensions(  
    out UINT Width  
) ;
```

Parameters

Width [out]

Type: [UINT](#)

The resource width, in texels.

Return value

Nothing

Remarks

This is a list of the overloaded versions of this method.

```
void GetDimensions (out UINT Width);  
  
void GetDimensions(out float Width);
```

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

See also

[RWTexture1D](#)

[Shader Model 5](#)

RWTexture1D::Load methods

Article • 11/06/2019 • 2 minutes to read

Reads texture data from a [RWTexture1D](#).

Overload list

Method	Description
Load(int,uint)	Reads texture data and returns status about the operation.
Load(int)	Reads texture data.

See also

[RWTexture1D](#)

RWTexture1D::Load(int) function

Article • 03/09/2021 • 2 minutes to read

Reads texture data.

Syntax

syntax

```
Load(  
    in int Location  
) ;
```

Parameters

Location [in]

Type: **int**

The location of the texture.

Return value

Type:

The return type matches the type in the declaration for the [RWTexture1D](#) object.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	x

See also

[Load methods](#)

RWTexture1D::Load(int,uint) function

Article • 03/09/2021 • 2 minutes to read

Reads texture data and returns status about the operation.

Syntax

syntax

```
Load(  
    in int Location,  
    out uint Status  
)
```

Parameters

Location [in]

Type: **int**

The location of the texture.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. **CheckAccessFullyMapped** returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, **CheckAccessFullyMapped** returns **FALSE**.

Return value

Type:

The return type matches the type in the declaration for the [RWTexture1D](#) object.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

See also

[Load methods](#)

RWTexture1D::Operator function

Article • 03/09/2021 • 2 minutes to read

Returns a resource variable.

Syntax

syntax

```
R Operator[](  
    in uint pos  
)
```

Parameters

pos [in]

Type: **uint**

The index position. Contains the x coordinate.

Return value

Type: **R**

A resource variable.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	x

See also

[RWTexture1D](#)

Shader Model 5

RWTexture1DArray

Article • 03/09/2021 • 2 minutes to read

A read/write resource.

Method	Description
GetDimensions	Gets the resource dimensions.
Load	Reads texture data.
Operator[]	Gets a resource variable.

You can prefix **RWTexture1DArray** objects with the storage class **globallycoherent**. This storage class causes memory barriers and syncs to flush data across the entire GPU such that other groups can see writes. Without this specifier, a memory barrier or sync will flush a UAV only within the current group.

A **RWTexture1DArray** object requires an element type in a declaration statement for the object. For example, the following declaration is correct:

```
RWTexture1DArray<float> tex;
```

Because a **RWTexture1DArray** object is a UAV-type object, its properties differ from a shader resource view (SRV)-type object, such as a **Texture1DArray** object. For example, you can read from and write to a **RWTexture1DArray** object, but you can only read from a **Texture1DArray** object.

A **RWTexture1DArray** object cannot use methods from a **Texture1DArray** object, such as **Sample**. However, because you can create multiple view types to the same resource, you can declare multiple texture types as a single texture in multiple shaders. For example, you can declare and use a **RWTexture1DArray** object as *tex* in a compute shader and then declare and use a **Texture1DArray** object as *tex* in a pixel shader.

ⓘ Note

The runtime enforces certain usage patterns when you create multiple view types to the same resource. For example, the runtime does not allow you to have both a

UAV mapping for a resource and SRV mapping for the same resource active at the same time.

Minimum Shader Model

This object is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This object is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

See also

[Shader Model 5 Objects](#)

RWTexture1DArray::GetDimensions function

Article • 03/09/2021 • 2 minutes to read

Returns the dimensions of the resource.

Syntax

syntax

```
void GetDimensions(  
    out UINT Width,  
    out UINT Elements  
) ;
```

Parameters

Width [out]

Type: [UINT](#)

The resource width, in texels.

Elements [out]

Type: [UINT](#)

The number of elements in the array.

Return value

This function does not return a value.

Remarks

This is a list of the overloaded versions of this method.

```
void GetDimensions (out UINT Width,  
    out UINT Elements);
```

```
void GetDimensions(out float Width,  
    out UINT Elements);
```

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

See also

[RWTexture1DArray](#)

[Shader Model 5](#)

RWTexture1DArray::Load methods

Article • 11/06/2019 • 2 minutes to read

Reads texture data from a [RWTexture1DArray](#).

Overload list

Method	Description
Load(int,uint)	Reads texture data and returns status about the operation.
Load(int)	Reads texture data.

See also

[RWTexture1DArray](#)

RWTexture1DArray::Load(int) function

Article • 03/09/2021 • 2 minutes to read

Reads texture data.

Syntax

syntax

```
Load(  
    in int Location  
)
```

Parameters

Location [in]

Type: **int**

The location of the texture.

Return value

Type:

The return type matches the type in the declaration for the [RWTexture1DArray](#) object.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	x

See also

[Load methods](#)

RWTexture1DArray::Load(int,uint) function

Article • 03/09/2021 • 2 minutes to read

Reads texture data and returns status about the operation.

Syntax

syntax

```
Load(  
    in int Location,  
    out uint Status  
)
```

Parameters

Location [in]

Type: **int**

The location of the texture.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. **CheckAccessFullyMapped** returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, **CheckAccessFullyMapped** returns **FALSE**.

Return value

Type:

The return type matches the type in the declaration for the [RWTexture1DArray](#) object.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

See also

[Load methods](#)

RWTexture1DArray::Operator function

Article • 03/09/2021 • 2 minutes to read

Returns a resource variable.

Syntax

syntax

```
R Operator[](  
    in uint2 pos  
)
```

Parameters

pos [in]

Type: `uint2`

The index position. The first component contains the x coordinate. The second component indicates the desired array slice.

Return value

Type: `R`

A resource variable.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	x

See also

RWTexture1DArray

Shader Model 5

RWTexture2D

Article • 03/09/2021 • 2 minutes to read

A read/write resource.

Method	Description
GetDimensions	Gets the resource dimensions.
Load	Reads texture data.
Operator[]	Gets a resource variable.

You can prefix RWTexture2D objects with the storage class **globallycoherent**. This storage class causes memory barriers and syncs to flush data across the entire GPU such that other groups can see writes. Without this specifier, a memory barrier or sync will flush only an unordered access view (UAV) within the current group.

A RWTexture2D object requires an element type in a declaration statement for the object. For example, the following declaration is not correct:

```
// The following declaration is incorrectly coded.  
RWTexture2D myTexture;
```

The following declaration is correct:

```
// The following declaration is correctly coded.  
RWTexture2D<float> tex;
```

Because a RWTexture2D object is a UAV-type object, its properties differ from a shader resource view (SRV)-type object, such as a [Texture2D](#) object. For example, you can read from and write to a RWTexture2D object, but you can only read from a Texture2D object.

A RWTexture2D object cannot use methods from a [Texture2D](#) object, such as [Sample](#). However, because you can create multiple view types to the same resource, you can declare multiple texture types as a single texture in multiple shaders. For example, the following code snippets show how you can declare and use a RWTexture2D object as *tex*

in a compute shader and then declare and use a `Texture2D` object as `tex` in a pixel shader.

ⓘ Note

The runtime enforces certain usage patterns when you create multiple view types to the same resource. For example, the runtime does not allow you to have both a UAV mapping for a resource and SRV mapping for the same resource active at the same time.

The following code is for the compute shader:

```
RWTexture2D<float> tex;
[numthreads(groupDim_x, groupDim_y, 1)]
void main(
    uint3 groupId : SV_GroupID,
    uint3 groupThreadId : SV_GroupThreadID,
    uint3 dispatchThreadId : SV_DispatchThreadID,
    uint groupIndex : SV_GroupIndex)
{
    tex [dispatchThreadId.xy] = <something>;
}
```

The following code is for the pixel shader:

```
struct PixelShaderInput
{
    float4 pos : SV_POSITION;
    float2 tex : TEXTURE;
};

Texture2D<float> tex;
float4 main(PixelShaderInput input) : SV_TARGET
{
    return tex.Sample(TextureSampler, input.tex);
}
```

Minimum Shader Model

This object is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This object is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

See also

[Shader Model 5 Objects](#)

RWTexture2D::GetDimensions function

Article • 03/09/2021 • 2 minutes to read

Returns the dimensions of the resource.

Syntax

syntax

```
void GetDimensions(  
    out UINT Width,  
    out UINT Height  
) ;
```

Parameters

Width [out]

Type: [UINT](#)

The resource width, in texels.

Height [out]

Type: [UINT](#)

The resource height, in texels.

Return value

This function does not return a value.

Remarks

This is a list of the overloaded versions of this method.

```
void GetDimensions (out UINT Width,  
    out UINT Height);
```

```
void GetDimensions(out float Width,  
    out float Height);
```

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

See also

[RWTexture2D](#)

[Shader Model 5](#)

RWTexture2D::Load methods

Article • 11/06/2019 • 2 minutes to read

Reads texture data from a [RWTexture2D](#).

Overload list

Method	Description
Load(int,uint)	Reads texture data and returns status about the operation.
Load(int)	Reads texture data.

See also

[RWTexture2D](#)

RWTexture2D::Load(int) function

Article • 03/09/2021 • 2 minutes to read

Reads texture data.

Syntax

syntax

```
Load(  
    in int Location  
) ;
```

Parameters

Location [in]

Type: **int**

The location of the texture.

Return value

Type:

The return type matches the type in the declaration for the [RWTexture2D](#) object.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	x

See also

[Load methods](#)

RWTexture2D::Load(int,uint) function

Article • 03/09/2021 • 2 minutes to read

Reads texture data and returns status about the operation.

Syntax

syntax

```
Load(  
    in int Location,  
    out uint Status  
)
```

Parameters

Location [in]

Type: **int**

The location of the texture.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. **CheckAccessFullyMapped** returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, **CheckAccessFullyMapped** returns **FALSE**.

Return value

Type:

The return type matches the type in the declaration for the [RWTexture2D](#) object.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

See also

[Load methods](#)

RWTexture2D::Operator function

Article • 03/09/2021 • 2 minutes to read

Returns a resource variable.

Syntax

syntax

```
R Operator[](  
    in uint2 pos  
)
```

Parameters

pos [in]

Type: **uint2**

The index position. Contains the (x, y) coordinates.

Return value

Type: **R**

A resource variable.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	x

See also

[RWTexture2D](#)

Shader Model 5

RWTexture2DArray

Article • 03/09/2021 • 2 minutes to read

A read/write resource.

Method	Description
GetDimensions	Gets the resource dimensions.
Load	Reads texture data.
Operator[]	Gets a resource variable.

You can prefix **RWTexture2DArray** objects with the storage class **globallycoherent**. This storage class causes memory barriers and syncs to flush data across the entire GPU such that other groups can see writes. Without this specifier, a memory barrier or sync will flush a UAV only within the current group.

A **RWTexture2DArray** object requires an element type in a declaration statement for the object. For example, the following declaration is correct:

```
RWTexture2DArray<float> tex;
```

Because a **RWTexture2DArray** object is a UAV-type object, its properties differ from a shader resource view (SRV)-type object, such as a **Texture2DArray** object. For example, you can read from and write to a **RWTexture2DArray** object, but you can only read from a **Texture2DArray** object.

A **RWTexture2DArray** object cannot use methods from a **Texture2DArray** object, such as [Sample](#). However, because you can create multiple view types to the same resource, you can declare multiple texture types as a single texture in multiple shaders. For example, you can declare and use a **RWTexture2DArray** object as *tex* in a compute shader and then declare and use a **Texture2DArray** object as *tex* in a pixel shader.

ⓘ Note

The runtime enforces certain usage patterns when you create multiple view types to the same resource. For example, the runtime does not allow you to have both a

UAV mapping for a resource and SRV mapping for the same resource active at the same time.

Minimum Shader Model

This object is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This object is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

See also

[Shader Model 5 Objects](#)

RWTexture2DArray::GetDimensions function

Article • 03/09/2021 • 2 minutes to read

Returns the dimensions of the resource.

Syntax

syntax

```
void GetDimensions(  
    out UINT Width,  
    out UINT Height,  
    out UINT Elements  
);
```

Parameters

Width [out]

Type: [UINT](#)

The resource width, in texels.

Height [out]

Type: [UINT](#)

The resource height, in texels.

Elements [out]

Type: [UINT](#)

The number of elements in the array.

Return value

This function does not return a value.

Remarks

This is a list of the overloaded versions of this method.

```
void GetDimensions (out UINT Width,  
                    out UINT Height,  
                    out UINT Elements);  
  
void GetDimensions(out float Width,  
                  out float Height,  
                  out float Elements);
```

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

See also

[RWTexture2DArray](#)

[Shader Model 5](#)

RWTexture2DArray::Load methods

Article • 11/06/2019 • 2 minutes to read

Reads texture data from a [RWTexture2DArray](#).

Overload list

Method	Description
Load(int,uint)	Reads texture data and returns status about the operation.
Load(int)	Reads texture data.

See also

[RWTexture2DArray](#)

RWTexture2DArray::Load(int) function

Article • 03/09/2021 • 2 minutes to read

Reads texture data.

Syntax

syntax

```
Load(  
    in int Location  
)
```

Parameters

Location [in]

Type: **int**

The location of the texture.

Return value

Type:

The return type matches the type in the declaration for the [RWTexture2DArray](#) object.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	x

See also

[Load methods](#)

RWTexture2DArray::Load(int,uint) function

Article • 03/09/2021 • 2 minutes to read

Reads texture data and returns status about the operation.

Syntax

syntax

```
Load(  
    in int Location,  
    out uint Status  
)
```

Parameters

Location [in]

Type: **int**

The location of the texture.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. **CheckAccessFullyMapped** returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, **CheckAccessFullyMapped** returns **FALSE**.

Return value

Type:

The return type matches the type in the declaration for the [RWTexture2DArray](#) object.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

See also

[Load methods](#)

RWTexture2DArray::Operator function

Article • 03/09/2021 • 2 minutes to read

Returns a resource variable.

Syntax

syntax

```
R Operator[](  
    in uint3 pos  
) ;
```

Parameters

pos [in]

Type: **uint3**

The index position. The first and second components contain the (x, y) coordinates. The third component indicates the desired array slice.

Return value

Type: **R**

A resource variable.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	x

See also

RWTexture2DArray

Shader Model 5

RWTexture3D

Article • 03/09/2021 • 2 minutes to read

A read/write resource.

Method	Description
GetDimensions	Gets the resource dimensions.
Load	Reads texture data.
Operator[]	Gets a resource variable.

You can prefix **RWTexture3D** objects with the storage class **globallycoherent**. This storage class causes memory barriers and syncs to flush data across the entire GPU such that other groups can see writes. Without this specifier, a memory barrier or sync will flush a UAV only within the current group.

A **RWTexture3D** object requires an element type in a declaration statement for the object. For example, the following declaration is correct:

```
RWTexture3D<float> tex;
```

Because a **RWTexture3D** object is a UAV-type object, its properties differ from a shader resource view (SRV)-type object, such as a **Texture3D** object. For example, you can read from and write to a **RWTexture3D** object, but you can only read from a **Texture3D** object.

A **RWTexture3D** object cannot use methods from a **Texture3D** object, such as [Sample](#). However, because you can create multiple view types to the same resource, you can declare multiple texture types as a single texture in multiple shaders. For example, you can declare and use a **RWTexture3D** object as *tex* in a compute shader and then declare and use a **Texture3D** object as *tex* in a pixel shader.

ⓘ Note

The runtime enforces certain usage patterns when you create multiple view types to the same resource. For example, the runtime does not allow you to have both a

UAV mapping for a resource and SRV mapping for the same resource active at the same time.

Minimum Shader Model

This object is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This object is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	x

See also

[Shader Model 5 Objects](#)

RWTexture3D::GetDimensions function

Article • 03/09/2021 • 2 minutes to read

Returns the dimensions of the resource.

Syntax

syntax

```
void GetDimensions(  
    out UINT Width,  
    out UINT Height,  
    out UINT Depth  
) ;
```

Parameters

Width [out]

Type: [UINT](#)

The resource width, in texels.

Height [out]

Type: [UINT](#)

The resource height, in texels.

Depth [out]

Type: [UINT](#)

The resource depth, in texels.

Return value

This function does not return a value.

Remarks

This is a list of the overloaded versions of this method.

```
void GetDimensions (out UINT Width,  
                    out UINT Height,  
                    out UINT Depth);
```

```
void GetDimensions(out float Width,  
                  out float Height,  
                  out float Depth);
```

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

See also

[RWTexture3D](#)

[Shader Model 5](#)

RWTexture3D::Load methods

Article • 11/06/2019 • 2 minutes to read

Reads texture data from a [RWTexture3D](#).

Overload list

Method	Description
Load(int,uint)	Reads texture data and returns status about the operation.
Load(int)	Reads texture data.

See also

[RWTexture3D](#)

RWTexture3D::Load(int) function

Article • 03/09/2021 • 2 minutes to read

Reads texture data.

Syntax

syntax

```
Load(  
    in int Location  
) ;
```

Parameters

Location [in]

Type: **int**

The location of the texture.

Return value

Type:

The return type matches the type in the declaration for the [RWTexture3D](#) object.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	x

See also

[Load methods](#)

RWTexture3D::Load(int,uint) function

Article • 03/09/2021 • 2 minutes to read

Reads texture data and returns status about the operation.

Syntax

syntax

```
Load(  
    in int Location,  
    out uint Status  
)
```

Parameters

Location [in]

Type: **int**

The location of the texture.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. **CheckAccessFullyMapped** returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, **CheckAccessFullyMapped** returns **FALSE**.

Return value

Type:

The return type matches the type in the declaration for the [RWTexture3D](#) object.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

See also

[Load methods](#)

RWTexture3D::Operator function

Article • 12/10/2020 • 2 minutes to read

Returns a resource variable of a [RWTexture3D](#).

Syntax

syntax

```
R Operator[](  
    in uint3 pos  
)
```

Parameters

pos [in]

Type: `uint3`

The index position. Contains the (x, y, z) coordinates.

Return value

Type: `R`

A resource variable.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	x

See also

[RWTexture3D](#)

Shader Model 5

StructuredBuffer

Article • 06/15/2022 • 2 minutes to read

A read-only buffer, which can take a T type that is a structure.

Method	Description
GetDimensions	Gets the resource dimensions.
Load	Reads buffer data.
Operator[]	Returns a read-only resource variable.

The SRV format bound to this resource needs to be created with the `DXGI_FORMAT_UNKNOWN` format.

To find out more about [structured buffers](#), see the overview material.

Minimum Shader Model

This object is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models Shader Model 4 (Available for compute and pixel shaders in Direct3D 11 on some Direct3D 10 devices.)	yes

This object is supported for the following types of shaders.

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Shader Model 5 Objects](#)

StructuredBuffer::GetDimensions function

Article • 03/09/2021 • 2 minutes to read

Gets the resource dimensions.

Syntax

syntax

```
void GetDimensions(  
    out uint numStructs,  
    out uint stride  
) ;
```

Parameters

numStructs [out]

Type: **uint**

The number of structures in the resource.

stride [out]

Type: **uint**

The stride, in bytes, of each structure element.

Return value

This function does not return a value.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[StructuredBuffer](#)

[Shader Model 5](#)

StructuredBuffer::Load methods

Article • 11/06/2019 • 2 minutes to read

Reads buffer data from a [StructuredBuffer](#).

Overload list

Method	Description
Load(int,uint)	Reads buffer data and returns status about the operation.
Load(int)	Reads buffer data.

See also

[StructuredBuffer](#)

StructuredBuffer::Load(int) function

Article • 03/09/2021 • 2 minutes to read

Reads buffer data.

Syntax

syntax

```
Load(  
    in int Location  
) ;
```

Parameters

Location [in]

Type: **int**

The location of the buffer.

Return value

Type:

The return type matches the type in the declaration for the [StructuredBuffer](#) object.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	x

See also

[Load methods](#)

StructuredBuffer::Load(int,uint) function

Article • 03/09/2021 • 2 minutes to read

Reads buffer data and returns status about the operation.

Syntax

syntax

```
Load(  
    in int Location,  
    out uint Status  
)
```

Parameters

Location [in]

Type: **int**

The location of the buffer.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. **CheckAccessFullyMapped** returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, **CheckAccessFullyMapped** returns **FALSE**.

Return value

Type:

The return type matches the type in the declaration for the [StructuredBuffer](#) object.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

See also

[Load methods](#)

StructuredBuffer::Operator function

Article • 12/10/2020 • 2 minutes to read

Returns a read-only resource variable of a [StructuredBuffer](#).

Syntax

syntax

```
R Operator[](  
    in uint pos  
)
```

Parameters

pos [in]

Type: **uint**

The index position.

Return value

Type: **R**

A read-only resource variable.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[StructuredBuffer](#)

Shader Model 5

Texture1D

Article • 10/24/2019 • 2 minutes to read

A 1D texture type ([as it exists in Shader Model 4](#)) plus resource variables. This texture object supports the following methods in addition to the methods in Shader Model 4.

Method	Description
GetDimensions	Gets the resource dimensions.
Load	Reads texture data.
Operator[]	Gets a read-only resource variable.
mips.Operator[][]	Gets a read-only resource variable.
Sample	Samples a texture.
SampleBias	Samples a texture, after applying the bias value to the mipmap level.
SampleCmp	Samples a texture, using a comparison value to reject samples.
SampleCmpLevelZero	Samples a texture (mipmap level 0 only), using a comparison value to reject samples.
SampleGrad	Samples a texture using a gradient to influence the way the sample location is calculated.
SampleLevel	Samples a texture on the specified mipmap level.

Minimum Shader Model

This object is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This object is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
--------	------	--------	----------	-------	---------

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

See also

[Shader Model 5 Objects](#)

Texture1D::GetDimensions function

Article • 03/09/2021 • 2 minutes to read

Returns the dimensions of the resource.

Syntax

syntax

```
void GetDimensions(  
    in  UINT MipLevel,  
    out UINT Width,  
    out UINT NumberOfLevels  
) ;
```

Parameters

MipLevel [in]

Type: [UINT](#)

Optional. Mipmap level (must be specified if *NumberOfLevels* is used).

Width [out]

Type: [UINT](#)

The resource width, in texels.

NumberOfLevels [out]

Type: [UINT](#)

The number of mipmap levels (requires *MipLevel* also).

Return value

This function does not return a value.

Remarks

This is a list of the overloaded versions of this method.

```
void GetDimensions(UINT MipLevel,  
    out UINT Width,  
    out UINT NumberOfLevels);  
  
void GetDimensions (out UINT Width);  
  
void GetDimensions(UINT MipLevel,  
    out float Width,  
    out float NumberOfLevels);  
  
void GetDimensions(out float Width);
```

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

See also

[Texture1D](#)

[Shader Model 5](#)

Texture1D::Load methods

Article • 11/06/2019 • 2 minutes to read

Reads texture data from a [Texture1D](#).

Overload list

Method	Description
Load(int,int)	Reads texture data.
Load(int,int,uint)	Reads texture data and returns status of the operation.

See also

[Texture1D](#)

Texture1D::Load(int,int,uint) function

Article • 03/09/2021 • 2 minutes to read

Reads texture data and returns status of the operation.

Syntax

syntax

```
Load(  
    in int Location,  
    in int Offset,  
    out uint Status  
) ;
```

Parameters

Location [in]

Type: **int**

The texture coordinates.

Offset [in]

Type: **int**

An offset applied to the texture coordinates before sampling.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns **TRUE** if all values from the corresponding `Sample`, `Gather`, or `Load` operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns **FALSE**.

Return value

Type:

The return type matches the type in the declaration for the [Texture1D](#) object.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

See also

[Load methods](#)

[Texture1D](#)

Texture1D::mips.Operator function

Article • 12/10/2020 • 2 minutes to read

Returns a read-only resource variable or a [Texture1D](#).

Syntax

syntax

```
R mips.Operator[][](  
    in uint mipSlice,  
    in uint pos  
)
```

Parameters

mipSlice [in]

Type: **uint**

The mip slice index.

pos [in]

Type: **uint**

The index position. Contains the x-coordinate.

Return value

Type: **R**

A read-only resource variable.

Remarks

Usage Example

```
Texture1D<float4> tex;  
uint mip = 2;  
uint pos = 1234;  
float4 f = tex.mips[mip][pos];
```

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

See also

[Texture1D](#)

[Shader Model 5](#)

Texture1D::Operator function

Article • 12/10/2020 • 2 minutes to read

Returns a read-only resource variable for a [Texture1D](#).

Syntax

syntax

```
R Operator[](  
    in uint pos  
)
```

Parameters

pos [in]

Type: **uint**

The index position. Contains the x-coordinate.

Return value

Type: **R**

A read-only resource variable.

Remarks

This method always accesses the first mip level. To specify other mip levels, use the [mip.operator\[\]\[\]](#) instead.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Texture1D](#)

[Shader Model 5](#)

Texture1D::Sample methods

Article • 11/06/2019 • 2 minutes to read

Samples a [Texture1D](#).

Overload list

Method	Description
Sample(S,float,int)	Samples a texture.
Sample(S,float,int,float)	Samples a texture with an optional value to clamp sample level-of-detail (LOD) values to.
Sample(S,float,int,float,uint)	Samples a texture with an optional value to clamp sample level-of-detail (LOD) values to, and returns status about the operation.

Remarks

Refer to [Texture-Object](#).

See also

[Texture1D](#)

Texture1D::Sample(S,float,int,float) function

Article • 03/09/2021 • 2 minutes to read

Samples a texture with an optional value to clamp sample level-of-detail (LOD) values to.

Syntax

Syntax

```
DXGI_FORMAT Sample(  
    in SamplerState S,  
    in float        Location,  
    in int         Offset,  
    in float        Clamp  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer miplevel; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[Sample methods](#)

Texture1D::Sample(S,float,int,float,uint) function

Article • 03/09/2021 • 2 minutes to read

Samples a texture with an optional value to clamp sample level-of-detail (LOD) values to, and returns status of the operation.

Syntax

Syntax

```
DXGI_FORMAT Sample(  
    in SamplerState S,  
    in float        Location,  
    in int         Offset,  
    in float        Clamp,  
    out uint       Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer miplevel; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns **TRUE** if all values from the corresponding [Sample](#), [Gather](#), or [Load](#) operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns **FALSE**.

Return value

Type: [**DXGI_FORMAT**](#)

The texture format, which is one of the typed values listed in [**DXGI_FORMAT**](#).

See also

[Sample methods](#)

Texture1D::SampleBias methods

Article • 11/06/2019 • 2 minutes to read

Samples a [Texture1D](#), after applying the bias value to the mipmap level.

Overload list

Method	Description
SampleBias(S,float,float,int)	Samples a texture, after applying the bias value to the mipmap level.
SampleBias(S,float,float,int,float)	Samples a texture, after applying the bias value to the mipmap level, with an optional value to clamp sample level-of-detail (LOD) values to.
SampleBias(S,float,float,int,float,uint)	Samples a texture, after applying the bias value to the mipmap level, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

Remarks

Refer to [Texture-Object](#).

See also

[Texture1D](#)

SampleBias::SampleBias(S,float,float,int,float) function for Texture1D

Article • 04/02/2021 • 2 minutes to read

Samples a texture, after applying the bias value to the mipmap level, with an optional value to clamp sample level-of-detail (LOD) values to.

Syntax

Syntax

```
DXGI_FORMAT SampleBias(  
    in SamplerState S,  
    in float        Location,  
    in float        Bias,  
    in int         Offset,  
    in float        Clamp  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

Bias [in]

Type: **float**

The bias value, which is a floating-point number between 0.0 and 1.0 inclusive, is applied to a mip level before sampling.

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer miplevel; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

SampleBias methods

SampleBias::SampleBias(S,float,float,int,float,uint) function for Texture1D

Article • 03/15/2021 • 2 minutes to read

Samples a texture, after applying the bias value to the mipmap level, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

Syntax

syntax

```
DXGI_FORMAT SampleBias(  
    in SamplerState S,  
    in float        Location,  
    in float        Bias,  
    in int         Offset,  
    in float        Clamp,  
    out uint       Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3

Texture-Object Type	Parameter Type
TextureCubeArray	float4

Bias [in]

Type: **float**

The bias value, which is a floating-point number between 0.0 and 1.0 inclusive, is applied to a mip level before sampling.

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer miplevel; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns

TRUE if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, **CheckAccessFullyMapped** returns FALSE.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[SampleBias methods](#)

Texture1D::SampleCmp methods

Article • 11/06/2019 • 2 minutes to read

Samples a [Texture1D](#), using a comparison value to reject samples.

Overload list

Method	Description
SampleCmp(S,float,float,int)	Samples a texture, using a comparison value to reject samples.
SampleCmp(S,float,float,int,float)	Samples a texture, using a comparison value to reject samples, with an optional value to clamp sample level-of-detail (LOD) values to.
SampleCmp(S,float,float,int,float,uint)	Samples a texture, using a comparison value to reject samples, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

Remarks

Refer to [Texture-Object](#).

See also

[Texture1D](#)

SampleCmp::SampleCmp(S,float,float,int,float) function for Texture1D

Article • 03/15/2021 • 2 minutes to read

Samples a texture, using a comparison value to reject samples, with an optional value to clamp sample level-of-detail (LOD) values to.

Syntax

Syntax

```
DXGI_FORMAT SampleCmp(  
    in SamplerState S,  
    in float        Location,  
    in float        CompareValue,  
    in int          Offset,  
    in float        Clamp  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

CompareValue [in]

Type: **float**

A floating-point value to use as a comparison value.

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer miplevel; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[SampleCmp methods](#)

SampleCmp::SampleCmp(S,float,float,int,float,uint) function for Texture1D

Article • 03/15/2021 • 2 minutes to read

Samples a texture, using a comparison value to reject samples, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

Syntax

Syntax

```
DXGI_FORMAT SampleCmp(  
    in SamplerState S,  
    in float        Location,  
    in float        CompareValue,  
    in int          Offset,  
    in float        Clamp,  
    out uint        Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3

Texture-Object Type	Parameter Type
TextureCubeArray	float4

CompareValue [in]

Type: **float**

A floating-point value to use as a comparison value.

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer mipmap level; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns TRUE if all values from the corresponding [Sample](#), [Gather](#), or [Load](#) operation accessed

mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns `FALSE`.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[SampleCmp methods](#)

Texture1D::SampleCmpLevelZero methods

Article • 11/06/2019 • 2 minutes to read

Samples a [Texture1D](#) on mipmap level 0 only and compares the result to a comparison value.

Overload list

Method	Description
SampleCmpLevelZero(S,float,float,int)	Samples a texture on mipmap level 0 only and compares the result to a comparison value.
SampleCmpLevelZero(S,float,float,int,uint)	Samples a texture on mipmap level 0 only and compares the result to a comparison value. Returns status about the operation.

Remarks

Refer to [Texture-Object](#).

See also

[Texture1D](#)

SampleCmpLevelZero::SampleCmpLevelZero(S,float,float,int,uint) function for Texture1D

Article • 03/15/2021 • 2 minutes to read

Samples a texture on mipmap level 0 only and compares the result to a comparison value. Returns status about the operation.

Syntax

```
syntax

DXGI_FORMAT SampleCmpLevelZero(
    in SamplerState S,
    in float          Location,
    in float          CompareValue,
    in int           Offset,
    out uint         Status
);
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3

Texture-Object Type	Parameter Type
TextureCubeArray	float4

CompareValue [in]

Type: **float**

A floating-point value to use as a comparison value.

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer mipmap level; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns **FALSE**.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[SampleCmpLevelZero methods](#)

Texture1D::SampleGrad methods

Article • 11/06/2019 • 2 minutes to read

Samples a [Texture1D](#) using a gradient to influence the way the sample location is calculated.

Overload list

Method	Description
SampleGrad(S,float,float,float,int)	Samples a texture, using a gradient to influence the way the sample location is calculated.
SampleGrad(S,float,float,float,int,float)	Samples a texture, using a gradient to influence the way the sample location is calculated, with an optional value to clamp sample level-of-detail (LOD) values to.
SampleGrad(S,float,float,float,int,float,uint)	Samples a texture, using a gradient to influence the way the sample location is calculated, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

Remarks

Refer to [Texture-Object](#).

See also

[Texture1D](#)

SampleGrad::SampleGrad(S,float,float, float,int,float) function for Texture1D

Article • 03/15/2021 • 2 minutes to read

Samples a texture, using a gradient to influence the way the sample location is calculated, with an optional value to clamp sample level-of-detail (LOD) values to.

Syntax

Syntax

```
DXGI_FORMAT SampleGrad(  
    in SamplerState S,  
    in float        Location,  
    in float        DDX,  
    in float        DDY,  
    in int          Offset,  
    in float        Clamp  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3

Texture-Object Type	Parameter Type
TextureCubeArray	float4

DDX [in]

Type: **float**

The rate of change of the surface geometry in the x direction. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	float
Texture2D, Texture2DArray	float2
Texture3D, TextureCube, TextureCubeArray	float3
Texture2DMS, Texture2DMSArray	not supported

DDY [in]

Type: **float**

The rate of change of the surface geometry in the y direction. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	float
Texture2D, Texture2DArray	float2
Texture3D, TextureCube, TextureCubeArray	float3
Texture2DMS, Texture2DMSArray	not supported

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer miplevel; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Return value

Type: **DXGI_FORMAT**

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[SampleGrad methods](#)

SampleGrad::SampleGrad(S,float,float,fl oat,int,float,uint) function for Texture1D

Article • 03/15/2021 • 2 minutes to read

Samples a texture, using a gradient to influence the way the sample location is calculated, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

Syntax

syntax

```
DXGI_FORMAT SampleGrad(  
    in SamplerState S,  
    in float        Location,  
    in float        DDX,  
    in float        DDY,  
    in int          Offset,  
    in float        Clamp,  
    out uint        Status  
) ;
```

Parameters

S [in]

Type: [SamplerState](#)

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: [float](#)

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2

Texture-Object Type	Parameter Type
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

DDX [in]

Type: **float**

The rate of change of the surface geometry in the x direction. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	float
Texture2D, Texture2DArray	float2
Texture3D, TextureCube, TextureCubeArray	float3
Texture2DMS, Texture2DMSArray	not supported

DDY [in]

Type: **float**

The rate of change of the surface geometry in the y direction. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	float
Texture2D, Texture2DArray	float2
Texture3D, TextureCube, TextureCubeArray	float3
Texture2DMS, Texture2DMSArray	not supported

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer miplevel; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns **TRUE** if all values from the corresponding [Sample](#), [Gather](#), or [Load](#) operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns **FALSE**.

Return value

Type: [**DXGI_FORMAT**](#)

The texture format, which is one of the typed values listed in [**DXGI_FORMAT**](#).

See also

[SampleGrad methods](#)

Texture1D::SampleLevel methods

Article • 11/06/2019 • 2 minutes to read

Samples a [Texture1D](#) on the specified mipmap level.

Overload list

Method	Description
SampleLevel(S,float,float,int)	Samples a texture on the specified mipmap level.
SampleLevel(S,float,float,int,uint)	Samples a texture on the specified mipmap level and returns status about the operation.

Remarks

Refer to [Texture-Object](#).

See also

[Texture1D](#)

SampleLevel::SampleLevel(S,float,float,int,uint) function for Texture1D

Article • 03/15/2021 • 2 minutes to read

Samples a texture on the specified mipmap level and returns status about the operation.

Syntax

syntax

```
DXGI_FORMAT SampleLevel(  
    in SamplerState S,  
    in float        Location,  
    in float        LOD,  
    in int         Offset,  
    out uint       Status  
)
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

LOD [in]

Type: **float**

[in] A number that specifies the mipmap level. If the value is ≤ 0 , mipmap level 0 (biggest map) is used. The fractional value (if supplied) is used to interpolate between two mipmap levels.

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer miplevel; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns **FALSE**.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[SampleLevel methods](#)

Texture1DArray

Article • 10/24/2019 • 2 minutes to read

Texture1DArray type (as it exists in Shader Model 4) plus resource variables. This texture object supports the following methods in addition to the methods in Shader Model 4.

Method	Description
GetDimensions	Gets the resource dimensions.
Load	Reads texture data.
mips.Operator[][]	Gets a read-only resource variable.
Operator[]	Gets a read-only resource variable.
Sample	Samples a texture.
SampleBias	Samples a texture, after applying the bias value to the mipmap level.
SampleCmp	Samples a texture, using a comparison value to reject samples.
SampleCmpLevelZero	Samples a texture (mipmap level 0 only), using a comparison value to reject samples.
SampleGrad	Samples a texture using a gradient to influence the way the sample location is calculated.
SampleLevel	Samples a texture on the specified mipmap level.

Minimum Shader Model

This object is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This object is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
--------	------	--------	----------	-------	---------

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

See also

[Shader Model 5 Objects](#)

Texture1DArray::GetDimensions function

Article • 03/09/2021 • 2 minutes to read

Returns the dimensions of the resource.

Syntax

syntax

```
void GetDimensions(  
    in  UINT MipLevel,  
    out UINT Width,  
    out UINT Elements,  
    out UINT NumberOfLevels  
) ;
```

Parameters

MipLevel [in]

Type: [UINT](#)

Optional. Mipmap level (must be specified if *NumberOfLevels* is used).

Width [out]

Type: [UINT](#)

The resource width, in texels.

Elements [out]

Type: [UINT](#)

The number of elements in the array.

NumberOfLevels [out]

Type: [UINT](#)

The number of mipmap levels (requires *MipLevel* also).

Return value

This function does not return a value.

Remarks

This is a list of the overloaded versions of this method.

```
void GetDimensions(UINT MipLevel,  
    out UINT Width,  
    out UINT Elements,  
    out UINT NumberOfLevels);  
  
void GetDimensions (out UINT Width,  
    out UINT Elements);  
  
void GetDimensions(UINT MipLevel,  
    out float Width,  
    out UINT Elements,  
    out float NumberOfLevels);  
  
void GetDimensions(out float Width,  
    out UINT Elements);
```

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

See also

[Texture1DArray](#)

[Shader Model 5](#)

Texture1DArray::Load methods

Article • 03/09/2021 • 2 minutes to read

Reads texture data.

Overload list

Method	Description
Load(int,int)	Reads texture data.
Load(int,int,uint)	Reads texture data and returns status of the operation.

See also

[Texture1DArray](#)

Texture1DArray::Load(int,int,uint) function

Article • 03/09/2021 • 2 minutes to read

Reads texture data and returns status of the operation.

Syntax

syntax

```
Load(  
    in int Location,  
    in int Offset,  
    out uint Status  
);
```

Parameters

Location [in]

Type: **int**

The texture coordinates.

Offset [in]

Type: **int**

An offset applied to the texture coordinates before sampling.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns **TRUE** if all values from the corresponding [Sample](#), [Gather](#), or [Load](#) operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns **FALSE**.

Return value

Type:

The return type matches the type in the declaration for the [Texture1DArray](#) object.

See also

[Load methods](#)

Texture1DArray::mips.Operator function

Article • 03/09/2021 • 2 minutes to read

Returns a read-only resource variable.

Syntax

syntax

```
R mips.Operator[][](  
    in uint mipSlice,  
    in uint2 pos  
)
```

Parameters

mipSlice [in]

Type: **uint**

The mip slice index.

pos [in]

Type: **uint2**

The index position. The first component contains the x-coordinate. The second component indicates the desired array slice.

Return value

Type: **R**

A read-only resource variable.

Remarks

Usage Example

```
Texture1DArray<float4> tex;  
uint mip = 2;  
uint2 pos_x_and_array = {1234, 3};  
float4 f = tex.mips[mip][pos_x_and_array];
```

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

See also

[Texture1DArray](#)

[Shader Model 5](#)

Texture1DArray::Operator function

Article • 03/09/2021 • 2 minutes to read

Returns a read-only resource variable.

Syntax

syntax

```
R Operator[](  
    in uint2 pos  
) ;
```

Parameters

pos [in]

Type: **uint2**

The index position. The first component contains the x-coordinate. The second component indicates the desired array slice.

Return value

Type: **R**

A read-only resource variable.

Remarks

This method always accesses the first mip level. To specify other mip levels, use the [mip.operator\[\]\(\)](#) method instead.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Texture1DArray](#)

[Shader Model 5](#)

Texture1DArray::Sample methods

Article • 03/09/2021 • 2 minutes to read

Samples a texture.

Overload list

Method	Description
Sample(S,float,int)	Samples a texture.
Sample(S,float,int,float)	Samples a texture with an optional value to clamp sample level-of-detail (LOD) values to.
Sample(S,float,int,float,uint)	Samples a texture with an optional value to clamp sample level-of-detail (LOD) values to, and returns status about the operation.

See also

[Texture1DArray](#)

[Texture-Object](#)

Texture1DArray::Sample(S,float,int,float) function

Article • 03/09/2021 • 2 minutes to read

Samples a texture with an optional value to clamp sample level-of-detail (LOD) values to.

Syntax

Syntax

```
DXGI_FORMAT Sample(  
    in SamplerState S,  
    in float        Location,  
    in int         Offset,  
    in float        Clamp  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer miplevel; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[Sample methods](#)

Texture1DArray::Sample(S,float,int,float,uint) function

Article • 03/09/2021 • 2 minutes to read

Samples a texture with an optional value to clamp sample level-of-detail (LOD) values to, and returns status of the operation.

Syntax

Syntax

```
DXGI_FORMAT Sample(  
    in SamplerState S,  
    in float        Location,  
    in int         Offset,  
    in float        Clamp,  
    out uint       Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer miplevel; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns **TRUE** if all values from the corresponding [Sample](#), [Gather](#), or [Load](#) operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns **FALSE**.

Return value

Type: [**DXGI_FORMAT**](#)

The texture format, which is one of the typed values listed in [**DXGI_FORMAT**](#).

See also

[Sample methods](#)

Texture1DArray::SampleBias methods

Article • 03/09/2021 • 2 minutes to read

Samples a texture, after applying the bias value to the mipmap level.

Overload list

Method	Description
SampleBias(S,float,float,int)	Samples a texture, after applying the bias value to the mipmap level.
SampleBias(S,float,float,int,float)	Samples a texture, after applying the bias value to the mipmap level, with an optional value to clamp sample level-of-detail (LOD) values to.
SampleBias(S,float,float,int,float,uint)	Samples a texture, after applying the bias value to the mipmap level, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

See also

[Texture1DArray](#)

[Texture-Object](#)

SampleBias::SampleBias(S,float,float,int,float) function for Texture1DArray

Article • 03/15/2021 • 2 minutes to read

Samples a texture, after applying the bias value to the mipmap level, with an optional value to clamp sample level-of-detail (LOD) values to.

Syntax

Syntax

```
DXGI_FORMAT SampleBias(  
    in SamplerState S,  
    in float        Location,  
    in float        Bias,  
    in int         Offset,  
    in float        Clamp  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

Bias [in]

Type: **float**

The bias value, which is a floating-point number between 0.0 and 1.0 inclusive, is applied to a mip level before sampling.

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer miplevel; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

SampleBias methods

SampleBias::SampleBias(S,float,float,int,float,uint) function for Texture1DArray

Article • 03/15/2021 • 2 minutes to read

Samples a texture, after applying the bias value to the mipmap level, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

Syntax

syntax

```
DXGI_FORMAT SampleBias(  
    in SamplerState S,  
    in float        Location,  
    in float        Bias,  
    in int         Offset,  
    in float        Clamp,  
    out uint       Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3

Texture-Object Type	Parameter Type
TextureCubeArray	float4

Bias [in]

Type: **float**

The bias value, which is a floating-point number between 0.0 and 1.0 inclusive, is applied to a mip level before sampling.

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer miplevel; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns

TRUE if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, **CheckAccessFullyMapped** returns FALSE.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[SampleBias methods](#)

Texture1DArray::SampleCmp methods

Article • 03/09/2021 • 2 minutes to read

Samples a texture, using a comparison value to reject samples.

Overload list

Method	Description
SampleCmp(S,float,float,int)	Samples a texture, using a comparison value to reject samples.
SampleCmp(S,float,float,int,float)	Samples a texture, using a comparison value to reject samples, with an optional value to clamp sample level-of-detail (LOD) values to.
SampleCmp(S,float,float,int,float,uint)	Samples a texture, using a comparison value to reject samples, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

See also

[Texture1DArray](#)

[Texture-Object](#)

SampleCmp::SampleCmp(S,float,float,int,float) function for Texture1DArray

Article • 03/15/2021 • 2 minutes to read

Samples a texture, using a comparison value to reject samples, with an optional value to clamp sample level-of-detail (LOD) values to.

Syntax

Syntax

```
DXGI_FORMAT SampleCmp(  
    in SamplerState S,  
    in float        Location,  
    in float        CompareValue,  
    in int          Offset,  
    in float        Clamp  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

CompareValue [in]

Type: **float**

A floating-point value to use as a comparison value.

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer miplevel; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[SampleCmp methods](#)

SampleCmp::SampleCmp(S,float,float,int,float,uint) function for Texture1DArray

Article • 03/15/2021 • 2 minutes to read

Samples a texture, using a comparison value to reject samples, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

Syntax

Syntax

```
DXGI_FORMAT SampleCmp(  
    in SamplerState S,  
    in float        Location,  
    in float        CompareValue,  
    in int          Offset,  
    in float        Clamp,  
    out uint        Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3

Texture-Object Type	Parameter Type
TextureCubeArray	float4

CompareValue [in]

Type: **float**

A floating-point value to use as a comparison value.

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer mipmap level; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns TRUE if all values from the corresponding [Sample](#), [Gather](#), or [Load](#) operation accessed

mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns `FALSE`.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[SampleCmp methods](#)

Texture1DArray::SampleCmpLevelZero methods

Article • 03/09/2021 • 2 minutes to read

Samples a texture on mipmap level 0 only, using a comparison value to reject samples.

Overload list

Method	Description
SampleCmpLevelZero(S,float,float,int)	Samples a texture on mipmap level 0 only and compares the result to a comparison value.
SampleCmpLevelZero(S,float,float,int,uint)	Samples a texture on mipmap level 0 only and compares the result to a comparison value. Returns status about the operation.

See also

[Texture1DArray](#)

[Texture-Object](#)

SampleCmpLevelZero::SampleCmpLevelZero(S,float,float,int,uint) function for Texture1DArray

Article • 03/15/2021 • 2 minutes to read

Samples a texture on mipmap level 0 only and compares the result to a comparison value. Returns status about the operation.

Syntax

```
syntax

DXGI_FORMAT SampleCmpLevelZero(
    in SamplerState S,
    in float          Location,
    in float          CompareValue,
    in int           Offset,
    out uint         Status
);
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3

Texture-Object Type	Parameter Type
TextureCubeArray	float4

CompareValue [in]

Type: **float**

A floating-point value to use as a comparison value.

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer mipmap level; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns **FALSE**.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[SampleCmpLevelZero methods](#)

Texture1DArray::SampleGrad methods

Article • 03/09/2021 • 2 minutes to read

Samples a texture using a gradient to influence the way the sample location is calculated.

Overload list

Method	Description
SampleGrad(S,float,float,float,int)	Samples a texture, using a gradient to influence the way the sample location is calculated.
SampleGrad(S,float,float,float,int,float)	Samples a texture, using a gradient to influence the way the sample location is calculated, with an optional value to clamp sample level-of-detail (LOD) values to.
SampleGrad(S,float,float,float,int,float,uint)	Samples a texture, using a gradient to influence the way the sample location is calculated, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

See also

[Texture1DArray](#)

[Texture-Object](#)

SampleGrad::SampleGrad(S,float,float,fl oat,int,float) function for Texture1DArray

Article • 03/15/2021 • 2 minutes to read

Samples a texture, using a gradient to influence the way the sample location is calculated, with an optional value to clamp sample level-of-detail (LOD) values to.

Syntax

Syntax

```
DXGI_FORMAT SampleGrad(  
    in SamplerState S,  
    in float        Location,  
    in float        DDX,  
    in float        DDY,  
    in int          Offset,  
    in float        Clamp  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3

Texture-Object Type	Parameter Type
TextureCubeArray	float4

DDX [in]

Type: **float**

The rate of change of the surface geometry in the x direction. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	float
Texture2D, Texture2DArray	float2
Texture3D, TextureCube, TextureCubeArray	float3
Texture2DMS, Texture2DMSArray	not supported

DDY [in]

Type: **float**

The rate of change of the surface geometry in the y direction. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	float
Texture2D, Texture2DArray	float2
Texture3D, TextureCube, TextureCubeArray	float3
Texture2DMS, Texture2DMSArray	not supported

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer miplevel; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Return value

Type: **DXGI_FORMAT**

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[SampleGrad methods](#)

SampleGrad::SampleGrad(S,float,float,fl oat,int,float,uint) function for Texture1DArray

Article • 03/15/2021 • 2 minutes to read

Samples a texture, using a gradient to influence the way the sample location is calculated, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

Syntax

syntax

```
DXGI_FORMAT SampleGrad(  
    in SamplerState S,  
    in float        Location,  
    in float        DDX,  
    in float        DDY,  
    in int          Offset,  
    in float        Clamp,  
    out uint        Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float

Texture-Object Type	Parameter Type
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

DDX [in]

Type: **float**

The rate of change of the surface geometry in the x direction. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	float
Texture2D, Texture2DArray	float2
Texture3D, TextureCube, TextureCubeArray	float3
Texture2DMS, Texture2DMSArray	not supported

DDY [in]

Type: **float**

The rate of change of the surface geometry in the y direction. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	float
Texture2D, Texture2DArray	float2
Texture3D, TextureCube, TextureCubeArray	float3
Texture2DMS, Texture2DMSArray	not supported

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer miplevel; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns TRUE if all values from the corresponding [Sample](#), [Gather](#), or [Load](#) operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns FALSE.

Return value

Type: [**DXGI_FORMAT**](#)

The texture format, which is one of the typed values listed in [**DXGI_FORMAT**](#).

See also

SampleGrad methods

Texture1DArray::SampleLevel methods

Article • 03/09/2021 • 2 minutes to read

Samples a texture on the specified mipmap level.

Overload list

Method	Description
SampleLevel(S,float,float,int)	Samples a texture on the specified mipmap level.
SampleLevel(S,float,float,int,uint)	Samples a texture on the specified mipmap level and returns status about the operation.

See also

[Texture1DArray](#)

[Texture-Object](#)

SampleLevel::SampleLevel(S,float,float,int,uint) function for Texture1DArray

Article • 03/15/2021 • 2 minutes to read

Samples a texture on the specified mipmap level and returns status about the operation.

Syntax

syntax

```
DXGI_FORMAT SampleLevel(  
    in SamplerState S,  
    in float        Location,  
    in float        LOD,  
    in int         Offset,  
    out uint       Status  
)
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

LOD [in]

Type: **float**

[in] A number that specifies the mipmap level. If the value is ≤ 0 , mipmap level 0 (biggest map) is used. The fractional value (if supplied) is used to interpolate between two mipmap levels.

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer miplevel; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns **FALSE**.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[SampleLevel methods](#)

Texture2D

Article • 02/12/2021 • 2 minutes to read

Texture2D type ([as it exists in Shader Model 4](#)) plus resource variables. This texture object supports the following methods in addition to the methods in Shader Model 4.

Method	Description
Gather	Returns the four texel values that would be used in a bi-linear filtering operation.
GatherRed	Returns the red components of the four texel values that would be used in a bi-linear filtering operation.
GatherGreen	Returns the green components of the four texel values that would be used in a bi-linear filtering operation.
GatherBlue	Returns the blue components of the four texel values that would be used in a bi-linear filtering operation.
GatherAlpha	Returns the alpha components of the four texel values that would be used in a bi-linear filtering operation.
GatherCmp	For four texel values that would be used in a bi-linear filtering operation, returns their comparison against a compare value.
GatherCmpRed	For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their red component against a compare value.
GatherCmpGreen	For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their green component against a compare value.
GatherCmpBlue	For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their blue component against a compare value.
GatherCmpAlpha	For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their alpha component against a compare value.
GetDimensions	Gets the resource dimensions.
Load	Reads texture data.
mips.Operator[][]	Gets a read-only resource variable.
Operator[]	Gets a read-only resource variable.
Sample	Samples a texture.
SampleBias	Samples a texture, after applying the bias value to the mipmap level.

Method	Description
SampleCmp	Samples a texture, using a comparison value to reject samples.
SampleCmpLevelZero	Samples a texture (mipmap level 0 only), using a comparison value to reject samples.
SampleGrad	Samples a texture using a gradient to influence the way the sample location is calculated.
SampleLevel	Samples a texture on the specified mipmap level.

Minimum Shader Model

This object is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This object is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Shader Model 5 Objects](#)

Texture2D::Gather methods

Article • 03/05/2021 • 2 minutes to read

Samples a [Texture2D](#) and returns all four components.

See the documentation on [gather4](#) for more information describing the underlying DXBC instruction.

Overload list

Method	Description
Gather(S,float,int)	Samples a texture and returns all four components.
Gather(S,float,int,uint)	Samples a texture and returns all four components along with status about the operation.

See also

[Texture2D](#)

Texture2D::Gather(S,float,int) function

Article • 03/09/2021 • 2 minutes to read

Returns the four texel values that would be used in a bi-linear filtering operation.

Syntax

syntax

```
TemplateType Gather(  
    in sampler s,  
    in float2 location,  
    in int2 offset  
) ;
```

Parameters

s [in]

Type: **sampler**

The zero-based sampler index.

location [in]

Type: **float2**

The sample coordinates (u,v).

offset [in]

Type: **int2**

The offset applied to the texture coordinates before sampling.

Return value

Type: **TemplateType**

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

See also

[Gather methods](#)

[Shader Model 5](#)

Texture2D::Gather(S,float,int,uint) function

Article • 03/06/2023 • 2 minutes to read

Returns the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

Syntax

Syntax

```
TemplateType Gather(  
    in  SamplerState S,  
    in  float2      Location,  
    in  int         Offset,  
    out uint       Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

Offset [in]

Type: **int**

The offset applied to the texture coordinates before sampling.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns **FALSE**.

Return value

Type: [TemplateType](#)

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Gather methods](#)

Texture2D::GatherAlpha methods

Article • 02/12/2021 • 2 minutes to read

Samples a [Texture2D](#) and returns the alpha component.

Overload list

Method	Description
GatherAlpha(S,float,int)	Returns the alpha components of the four texel values that would be used in a bi-linear filtering operation.
GatherAlpha(S,float,int,uint)	Returns the alpha components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.
GatherAlpha(S,float,int2,int2,int2,int2)	Returns the alpha components of the four texel values that would be used in a bi-linear filtering operation.
GatherAlpha(S,float,int2,int2,int2,int2,uint)	Returns the alpha components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

See also

[Texture2D](#)

Texture2D::GatherAlpha(S,float,int) function

Article • 03/09/2021 • 2 minutes to read

Returns the alpha components of the four texel values that would be used in a bi-linear filtering operation.

Syntax

Syntax

```
TemplateType GatherAlpha(  
    in sampler s,  
    in float2 location,  
    in int2 offset  
) ;
```

Parameters

s [in]

Type: **sampler**

The zero-based sampler index.

location [in]

Type: **float2**

The sample coordinates (u,v).

offset [in]

Type: **int2**

An offset that is applied to the texture coordinate before sampling.

Return value

Type: **TemplateType**

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherAlpha methods](#)

[Shader Model 5](#)

Texture2D::GatherAlpha(S,float,int2,int2,int2,int2) function

Article • 03/06/2023 • 2 minutes to read

Returns the alpha components of the four texel values that would be used in a bi-linear filtering operation.

Syntax

Syntax

```
TemplateType GatherAlpha(  
    in SamplerState S,  
    in float2      Location,  
    in int2       Offset1,  
    in int2       Offset2,  
    in int2       Offset3,  
    in int2       Offset4  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

Offset1 [in]

Type: **int2**

The first offset component applied to the texture coordinates before sampling.

Offset2 [in]

Type: **int2**

The second offset component applied to the texture coordinates before sampling.

Offset3 [in]

Type: **int2**

The third offset component applied to the texture coordinates before sampling.

Offset4 [in]

Type: **int2**

The fourth offset component applied to the texture coordinates before sampling.

Return value

Type: **TemplateType**

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherAlpha methods](#)

Texture2D::GatherAlpha(S,float,int,uint) function

Article • 03/06/2023 • 2 minutes to read

Returns the alpha components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

Syntax

Syntax

```
TemplateType GatherAlpha(  
    in  SamplerState S,  
    in  float2      Location,  
    in  int         Offset,  
    out uint       Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

Offset [in]

Type: **int**

The offset applied to the texture coordinates before sampling.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns **FALSE**.

Return value

Type: [TemplateType](#)

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherAlpha methods](#)

Texture2D::GatherAlpha(S,float,int2,int2,int2,int2,uint) function

Article • 03/06/2023 • 2 minutes to read

Returns the alpha components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

Syntax

Syntax

```
TemplateType GatherAlpha(  
    in SamplerState S,  
    in float2      Location,  
    in int2        Offset1,  
    in int2        Offset2,  
    in int2        Offset3,  
    in int2        Offset4,  
    out uint       Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

Offset1 [in]

Type: **int2**

The first offset component applied to the texture coordinates before sampling.

Offset2 [in]

Type: **int2**

The second offset component applied to the texture coordinates before sampling.

Offset3 [in]

Type: **int2**

The third offset component applied to the texture coordinates before sampling.

Offset4 [in]

Type: **int2**

The fourth offset component applied to the texture coordinates before sampling.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. **CheckAccessFullyMapped** returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, **CheckAccessFullyMapped** returns **FALSE**.

Return value

Type: **TemplateType**

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherAlpha methods](#)

Texture2D::GatherBlue methods

Article • 02/12/2021 • 2 minutes to read

Samples a [Texture2D](#) and returns the blue component.

Overload list

Method	Description
GatherBlue(S,float,int)	Returns the blue components of the four texel values that would be used in a bi-linear filtering operation.
GatherBlue(S,float,int,uint)	Returns the blue components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.
GatherBlue(S,float,int2,int2,int2,int2)	Returns the blue components of the four texel values that would be used in a bi-linear filtering operation.
GatherBlue(S,float,int2,int2,int2,int2,uint)	Returns the blue components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

See also

[Texture2D](#)

Texture2D::GatherBlue(S,float,int) function

Article • 03/09/2021 • 2 minutes to read

Returns the blue components of the four texel values that would be used in a bi-linear filtering operation.

Syntax

Syntax

```
TemplateType GatherBlue(  
    in sampler s,  
    in float2 location,  
    in int2 offset  
) ;
```

Parameters

s [in]

Type: **sampler**

The zero-based sampler index.

location [in]

Type: **float2**

The sample coordinates (u,v).

offset [in]

Type: **int2**

An offset that is applied to the texture coordinate before sampling.

Return value

Type: **TemplateType**

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherBlue methods](#)

[Shader Model 5](#)

GatherBlue(S,float,int2,int2,int2,int2) function

Article • 03/06/2023 • 2 minutes to read

Returns the blue components of the four texel values that would be used in a bi-linear filtering operation.

Syntax

Syntax

```
TemplateType GatherBlue(  
    in SamplerState S,  
    in float2      Location,  
    in int2       Offset1,  
    in int2       Offset2,  
    in int2       Offset3,  
    in int2       Offset4  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

Offset1 [in]

Type: **int2**

The first offset component applied to the texture coordinates before sampling.

Offset2 [in]

Type: **int2**

The second offset component applied to the texture coordinates before sampling.

Offset3 [in]

Type: `int2`

The third offset component applied to the texture coordinates before sampling.

Offset4 [in]

Type: `int2`

The fourth offset component applied to the texture coordinates before sampling.

Return value

Type: `TemplateType`

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherBlue methods](#)

Texture2D::GatherBlue(S,float,int,uint) function

Article • 03/06/2023 • 2 minutes to read

Returns the blue components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

Syntax

Syntax

```
TemplateType GatherBlue(  
    in  SamplerState S,  
    in  float2      Location,  
    in  int         Offset,  
    out uint       Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

Offset [in]

Type: **int**

The offset applied to the texture coordinates before sampling.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns **FALSE**.

Return value

Type: [TemplateType](#)

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherBlue methods](#)

Texture2D::GatherBlue(S,float,int2,int2,int2,int2,uint) function

Article • 03/06/2023 • 2 minutes to read

Returns the blue components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

Syntax

Syntax

```
TemplateType GatherBlue(  
    in SamplerState S,  
    in float      Location,  
    in int2       Offset1,  
    in int2       Offset2,  
    in int2       Offset3,  
    in int2       Offset4,  
    out uint      Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

Offset1 [in]

Type: **int2**

The first offset component applied to the texture coordinates before sampling.

Offset2 [in]

Type: **int2**

The second offset component applied to the texture coordinates before sampling.

Offset3 [in]

Type: `int2`

The third offset component applied to the texture coordinates before sampling.

Offset4 [in]

Type: `int2`

The fourth offset component applied to the texture coordinates before sampling.

Status [out]

Type: `uint`

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns `TRUE` if all values from the corresponding `Sample`, `Gather`, or `Load` operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns `FALSE`.

Return value

Type: `TemplateType`

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherBlue methods](#)

Texture2D::GatherCmp methods

Article • 03/05/2021 • 2 minutes to read

For four texel values of a [Texture2D](#) that would be used in a bi-linear filtering operation, returns their comparison against a compare value.

See the documentation on [gather4_c](#) for more information describing the underlying DXBC instruction.

Overload list

Method	Description
GatherCmp(S,float,float,int)	Samples and compares a texture and returns all four components.
GatherCmp(S,float,float,int,uint)	Samples and compares a texture and returns all four components along with status about the operation.

See also

[Texture2D](#)

Texture2D::GatherCmp(S,float,float,int) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns their comparison against a compare value.

Syntax

Syntax

```
float4 GatherCmp(  
    in SamplerComparisonState s,  
    in float2 location,  
    in float compare_value,  
    in int2 offset  
) ;
```

Parameters

s [in]

Type: **SamplerComparisonState**

The zero-based sampler index.

location [in]

Type: **float2**

The sample coordinates (u,v).

compare_value [in]

Type: **float**

A value to compare each against each sampled value.

offset [in]

Type: **int2**

An offset that is applied to the texture coordinate before sampling.

Return value

Type: **float4**

A four-component value, each component is the result of a per-component comparison.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmp methods](#)

[Shader Model 5](#)

Texture2D::GatherCmp(S,float,float,int,uint) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns their comparison against a compare value along with tile-mapping status.

Syntax

Syntax

```
TemplateType GatherCmp(  
    in SamplerState S,  
    in float        Location,  
    in float        CompareValue,  
    in int2         Offset,  
    out uint        Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

CompareValue [in]

Type: **float**

A value to compare each against each sampled value.

Offset [in]

Type: **int2**

The offset in texels applied to the texture coordinates before sampling. Must be a literal value.

Status [out]

Type: `uint`

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns `TRUE` if all values from the corresponding `Sample`, `Gather`, or `Load` operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns `FALSE`.

Return value

Type: `TemplateType`

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmp methods](#)

Texture2D::GatherCmpAlpha methods

Article • 11/06/2019 • 2 minutes to read

Samples and compares a [Texture2D](#) and returns the alpha component.

Overload list

Method	Description
GatherCmpAlpha(S,float,float,int)	Samples and compares a texture and returns the alpha component.
GatherCmpAlpha(S,float,float,int,uint)	Samples and compares a texture and returns the alpha component along with status about the operation.
GatherCmpAlpha(S,float,float,int2,int2,int2,int2)	Samples and compares a texture and returns the alpha component.
GatherCmpAlpha(S,float,float,int2,int2,int2,int2,uint)	Samples and compares a texture and returns the alpha component along with status about the operation.

See also

[Texture2D](#)

Texture2D::GatherCmpAlpha(S,float,float,int) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their alpha component against a compare value.

Syntax

Syntax

```
float4 GatherCmpAlpha(  
    in SamplerComparisonState s,  
    in float2 location,  
    in float compare_value,  
    in int2 offset  
) ;
```

Parameters

s [in]

Type: **SamplerComparisonState**

The zero-based sampler index.

location [in]

Type: **float2**

The sample coordinates (u,v).

compare_value [in]

Type: **float**

A value to compare each against each sampled value.

offset [in]

Type: **int2**

An offset that is applied to the texture coordinate before sampling.

Return value

Type: **float4**

A four-component value, each component is the result of a per-component comparison.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmpAlpha methods](#)

[Shader Model 5](#)

Texture2D::GatherCmpAlpha(S,float,floa t,int2,int2,int2,int2) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their alpha component against a compare value.

Syntax

Syntax

```
TemplateType GatherCmpAlpha(  
    in SamplerState S,  
    in float      Location,  
    in float      CompareValue,  
    in int2       Offset1,  
    in int2       Offset2,  
    in int2       Offset3,  
    in int2       Offset4  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

CompareValue [in]

Type: **float**

A value to compare each against each sampled value.

Offset1 [in]

Type: **int2**

The first offset component applied to the texture coordinates before sampling.

Offset2 [in]

Type: **int2**

The second offset component applied to the texture coordinates before sampling.

Offset3 [in]

Type: **int2**

The third offset component applied to the texture coordinates before sampling.

Offset4 [in]

Type: **int2**

The fourth offset component applied to the texture coordinates before sampling.

Return value

Type: **TemplateType**

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmpAlpha methods](#)

Texture2D::GatherCmpAlpha(S,float,floa t,int,uint) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their alpha component against a compare value along with tile-mapping status.

Syntax

syntax

```
TemplateType GatherCmpAlpha(  
    in SamplerState S,  
    in float        Location,  
    in float        CompareValue,  
    in int          Offset,  
    out uint        Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

CompareValue [in]

Type: **float**

A value to compare each against each sampled value.

Offset [in]

Type: **int**

The offset applied to the texture coordinates before sampling.

Status [out]

Type: `uint`

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns `TRUE` if all values from the corresponding `Sample`, `Gather`, or `Load` operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns `FALSE`.

Return value

Type: `TemplateType`

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmpAlpha methods](#)

Texture2D::GatherCmpAlpha(S,float,floa t,int2,int2,int2,int2,uint) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their alpha component against a compare value along with tile-mapping status.

Syntax

syntax

```
TemplateType GatherCmpAlpha(  
    in SamplerState S,  
    in float        Location,  
    in float        CompareValue,  
    in int2         Offset1,  
    in int2         Offset2,  
    in int2         Offset3,  
    in int2         Offset4,  
    out uint        Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

CompareValue [in]

Type: **float**

A value to compare each against each sampled value.

Offset1 [in]

Type: **int2**

The first offset component applied to the texture coordinates before sampling.

Offset2 [in]

Type: **int2**

The second offset component applied to the texture coordinates before sampling.

Offset3 [in]

Type: **int2**

The third offset component applied to the texture coordinates before sampling.

Offset4 [in]

Type: **int2**

The fourth offset component applied to the texture coordinates before sampling.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. **CheckAccessFullyMapped** returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, **CheckAccessFullyMapped** returns **FALSE**.

Return value

Type: **TemplateType**

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
--------	------	--------	----------	-------	---------

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

See also

[GatherCmpAlpha methods](#)

Texture2D::GatherCmpBlue methods

Article • 11/06/2019 • 2 minutes to read

Samples and compares a [Texture2D](#) and returns the blue component.

Overload list

Method	Description
GatherCmpBlue(S,float,float,int)	Samples and compares a texture and returns the blue component.
GatherCmpBlue(S,float,float,int,uint)	Samples and compares a texture and returns the blue component along with status about the operation.
GatherCmpBlue(S,float,float,int2,int2,int2,int2)	Samples and compares a texture and returns the blue component.
GatherCmpBlue(S,float,float,int2,int2,int2,int2,uint)	Samples and compares a texture and returns the blue component along with status about the operation.

See also

[Texture2D](#)

Texture2D::GatherCmpBlue(S,float,float,int) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their blue component against a compare value.

Syntax

Syntax

```
float4 GatherCmpBlue(  
    in SamplerComparisonState s,  
    in float2 location,  
    in float compare_value,  
    in int2 offset  
) ;
```

Parameters

s [in]

Type: **SamplerComparisonState**

The zero-based sampler index.

location [in]

Type: **float2**

The sample coordinates (u,v).

compare_value [in]

Type: **float**

A value to compare each against each sampled value.

offset [in]

Type: **int2**

An offset that is applied to the texture coordinate before sampling.

Return value

Type: **float4**

A four-component value, each component is the result of a per-component comparison.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmpBlue methods](#)

[Shader Model 5](#)

Texture2D::GatherCmpBlue(S,float,float,int2,int2,int2,int2) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their blue component against a compare value.

Syntax

Syntax

```
TemplateType GatherCmpBlue(
    in SamplerState S,
    in float      Location,
    in float      CompareValue,
    in int2       Offset1,
    in int2       Offset2,
    in int2       Offset3,
    in int2       Offset4
);
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

CompareValue [in]

Type: **float**

A value to compare each against each sampled value.

Offset1 [in]

Type: **int2**

The first offset component applied to the texture coordinates before sampling.

Offset2 [in]

Type: **int2**

The second offset component applied to the texture coordinates before sampling.

Offset3 [in]

Type: **int2**

The third offset component applied to the texture coordinates before sampling.

Offset4 [in]

Type: **int2**

The fourth offset component applied to the texture coordinates before sampling.

Return value

Type: **TemplateType**

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmpBlue methods](#)

Texture2D::GatherCmpBlue(S,float,float,int,uint) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their blue component against a compare value.

Syntax

Syntax

```
TemplateType GatherCmpBlue(
    in SamplerState S,
    in float        Location,
    in float        CompareValue,
    in int         Offset,
    out uint       Status
);
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

CompareValue [in]

Type: **float**

A value to compare each against each sampled value.

Offset [in]

Type: **int**

The offset applied to the texture coordinates before sampling.

Status [out]

Type: `uint`

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns `TRUE` if all values from the corresponding `Sample`, `Gather`, or `Load` operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns `FALSE`.

Return value

Type: `TemplateType`

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmpBlue methods](#)

Texture2D::GatherCmpBlue(S,float,float,int2,int2,int2,uint) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their blue component against a compare value along with tile-mapping status.

Syntax

syntax

```
TemplateType GatherCmpBlue(
    in SamplerState S,
    in float        Location,
    in float        CompareValue,
    in int2         Offset1,
    in int2         Offset2,
    in int2         Offset3,
    in int2         Offset4,
    out uint        Status
);
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

CompareValue [in]

Type: **float**

A value to compare each against each sampled value.

Offset1 [in]

Type: **int2**

The first offset component applied to the texture coordinates before sampling.

Offset2 [in]

Type: **int2**

The second offset component applied to the texture coordinates before sampling.

Offset3 [in]

Type: **int2**

The third offset component applied to the texture coordinates before sampling.

Offset4 [in]

Type: **int2**

The fourth offset component applied to the texture coordinates before sampling.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. **CheckAccessFullyMapped** returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, **CheckAccessFullyMapped** returns **FALSE**.

Return value

Type: **TemplateType**

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
--------	------	--------	----------	-------	---------

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

See also

[GatherCmpBlue methods](#)

Texture2D::GatherCmpGreen methods

Article • 11/06/2019 • 2 minutes to read

Samples and compares a [Texture2D](#) and returns the green component.

Overload list

Method	Description
GatherCmpGreen(S,float,float,int)	Samples and compares a texture and returns the green component.
GatherCmpGreen(S,float,float,int,uint)	Samples and compares a texture and returns the green component along with status about the operation.
GatherCmpGreen(S,float,float,int2,int2,int2,int2)	Samples and compares a texture and returns the green component.
GatherCmpGreen(S,float,float,int2,int2,int2,int2,uint)	Samples and compares a texture and returns the green component along with status about the operation.

See also

[Texture2D](#)

Texture2D::GatherCmpGreen(S,float,float,int) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their green component against a compare value.

Syntax

Syntax

```
float4 GatherCmpGreen(  
    in SamplerComparisonState s,  
    in float2 location,  
    in float compare_value,  
    in int2 offset  
) ;
```

Parameters

s [in]

Type: **SamplerComparisonState**

The zero-based sampler index.

location [in]

Type: **float2**

The sample coordinates (u,v).

compare_value [in]

Type: **float**

A value to compare each against each sampled value.

offset [in]

Type: **int2**

An offset that is applied to the texture coordinate before sampling.

Return value

Type: **float4**

A four-component value, each component is the result of a per-component comparison.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmpGreen methods](#)

[Shader Model 5](#)

Texture2D::GatherCmpGreen(S,float,floa t,int2,int2,int2,int2) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their green component against a compare value.

Syntax

Syntax

```
TemplateType GatherCmpGreen(  
    in SamplerState S,  
    in float      Location,  
    in float      CompareValue,  
    in int2       Offset1,  
    in int2       Offset2,  
    in int2       Offset3,  
    in int2       Offset4  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

CompareValue [in]

Type: **float**

A value to compare each against each sampled value.

Offset1 [in]

Type: **int2**

The first offset component applied to the texture coordinates before sampling.

Offset2 [in]

Type: **int2**

The second offset component applied to the texture coordinates before sampling.

Offset3 [in]

Type: **int2**

The third offset component applied to the texture coordinates before sampling.

Offset4 [in]

Type: **int2**

The fourth offset component applied to the texture coordinates before sampling.

Return value

Type: **TemplateType**

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmpGreen methods](#)

Texture2D::GatherCmpGreen(S,float,floa t,int,uint) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their green component against a compare value along with tile-mapping status.

Syntax

syntax

```
TemplateType GatherCmpGreen(  
    in SamplerState S,  
    in float        Location,  
    in float        CompareValue,  
    in int          Offset,  
    out uint        Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

CompareValue [in]

Type: **float**

A value to compare each against each sampled value.

Offset [in]

Type: **int**

The offset applied to the texture coordinates before sampling.

Status [out]

Type: `uint`

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns `TRUE` if all values from the corresponding `Sample`, `Gather`, or `Load` operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns `FALSE`.

Return value

Type: `TemplateType`

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmpGreen methods](#)

Texture2D::GatherCmpGreen(S,float,floa t,int2,int2,int2,int2,uint) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their green component against a compare value along with tile-mapping status.

Syntax

syntax

```
TemplateType GatherCmpGreen(  
    in SamplerState S,  
    in float      Location,  
    in float      CompareValue,  
    in int2       Offset1,  
    in int2       Offset2,  
    in int2       Offset3,  
    in int2       Offset4,  
    out uint     Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

CompareValue [in]

Type: **float**

A value to compare each against each sampled value.

Offset1 [in]

Type: **int2**

The first offset component applied to the texture coordinates before sampling.

Offset2 [in]

Type: **int2**

The second offset component applied to the texture coordinates before sampling.

Offset3 [in]

Type: **int2**

The third offset component applied to the texture coordinates before sampling.

Offset4 [in]

Type: **int2**

The fourth offset component applied to the texture coordinates before sampling.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. **CheckAccessFullyMapped** returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, **CheckAccessFullyMapped** returns **FALSE**.

Return value

Type: **TemplateType**

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
--------	------	--------	----------	-------	---------

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

See also

[GatherCmpGreen methods](#)

Texture2D::GatherCmpRed methods

Article • 11/06/2019 • 2 minutes to read

Samples and compares a [Texture2D](#) and returns the red component.

Overload list

Method	Description
GatherCmpRed(S,float,float,int)	Samples and compares a texture and returns the red component.
GatherCmpRed(S,float,float,int,uint)	Samples and compares a texture and returns the red component along with status about the operation.
GatherCmpRed(S,float,float,int2,int2,int2,int2)	Samples and compares a texture and returns the red component.
GatherCmpRed(S,float,float,int2,int2,int2,int2,uint)	Samples and compares a texture and returns the red component along with status about the operation.

See also

[Texture2D](#)

Texture2D::GatherCmpRed(S,float,float,int) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their red component against a compare value.

Syntax

Syntax

```
float4 GatherCmpRed(  
    in SamplerComparisonState s,  
    in float2 location,  
    in float compare_value,  
    in int2 offset  
) ;
```

Parameters

s [in]

Type: **SamplerComparisonState**

The zero-based sampler index.

location [in]

Type: **float2**

The sample coordinates (u,v).

compare_value [in]

Type: **float**

A value to compare each against each sampled value.

offset [in]

Type: **int2**

An offset that is applied to the texture coordinate before sampling.

Return value

Type: **float4**

A four-component value, each component is the result of a per-component comparison.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmpRed methods](#)

[Shader Model 5](#)

Texture2D::GatherCmpRed(S,float,float,int2,int2,int2,int2) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their red component against a compare value.

Syntax

Syntax

```
TemplateType GatherCmpRed(  
    in SamplerState S,  
    in float      Location,  
    in float      CompareValue,  
    in int2       Offset1,  
    in int2       Offset2,  
    in int2       Offset3,  
    in int2       Offset4  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

CompareValue [in]

Type: **float**

A value to compare each against each sampled value.

Offset1 [in]

Type: **int2**

The first offset component applied to the texture coordinates before sampling.

Offset2 [in]

Type: **int2**

The second offset component applied to the texture coordinates before sampling.

Offset3 [in]

Type: **int2**

The third offset component applied to the texture coordinates before sampling.

Offset4 [in]

Type: **int2**

The fourth offset component applied to the texture coordinates before sampling.

Return value

Type: **TemplateType**

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmpRed methods](#)

Texture2D::GatherCmpRed(S,float,float,int,uint) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their red component against a compare value along with tile-mapping status.

Syntax

syntax

```
TemplateType GatherCmpRed(
    in SamplerState S,
    in float        Location,
    in float        CompareValue,
    in int         Offset,
    out uint       Status
);
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

CompareValue [in]

Type: **float**

A value to compare each against each sampled value.

Offset [in]

Type: **int**

The offset applied to the texture coordinates before sampling.

Status [out]

Type: `uint`

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns `TRUE` if all values from the corresponding `Sample`, `Gather`, or `Load` operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns `FALSE`.

Return value

Type: `TemplateType`

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmpRed methods](#)

Texture2D::GatherCmpRed(S,float,float,int2,int2,int2,uint) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their red component against a compare value along with tile-mapping status.

Syntax

syntax

```
TemplateType GatherCmpRed(
    in SamplerState S,
    in float        Location,
    in float        CompareValue,
    in int2         Offset1,
    in int2         Offset2,
    in int2         Offset3,
    in int2         Offset4,
    out uint        Status
);
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

CompareValue [in]

Type: **float**

A value to compare each against each sampled value.

Offset1 [in]

Type: **int2**

The first offset component applied to the texture coordinates before sampling.

Offset2 [in]

Type: **int2**

The second offset component applied to the texture coordinates before sampling.

Offset3 [in]

Type: **int2**

The third offset component applied to the texture coordinates before sampling.

Offset4 [in]

Type: **int2**

The fourth offset component applied to the texture coordinates before sampling.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. **CheckAccessFullyMapped** returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, **CheckAccessFullyMapped** returns **FALSE**.

Return value

Type: **TemplateType**

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
--------	------	--------	----------	-------	---------

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

See also

[GatherCmpRed methods](#)

Texture2D::GatherGreen methods

Article • 02/12/2021 • 2 minutes to read

Samples a [Texture2D](#) and returns the green component.

Overload list

Method	Description
GatherGreen(S,float,int)	Returns the green components of the four texel values that would be used in a bi-linear filtering operation.
GatherGreen(S,float,int,uint)	Returns the green components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.
GatherGreen(S,float,int2,int2,int2,int2)	Returns the green components of the four texel values that would be used in a bi-linear filtering operation.
GatherGreen(S,float,int2,int2,int2,int2,uint)	Returns the green components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

See also

[Texture2D](#)

Texture2D::GatherGreen(S,float,int) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their green component against a compare value.

Syntax

Syntax

```
TemplateType GatherGreen(  
    in sampler s,  
    in float2 location,  
    in int2 offset  
) ;
```

Parameters

s [in]

Type: **sampler**

The zero-based sampler index.

location [in]

Type: **float2**

The sample coordinates (u,v).

offset [in]

Type: **int2**

An offset that is applied to the texture coordinate before sampling.

Return value

Type: **TemplateType**

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherGreen methods](#)

[Shader Model 5](#)

Texture2D::GatherGreen(S,float,int2,int2,int2,int2) function

Article • 03/06/2023 • 2 minutes to read

Returns the green components of the four texel values that would be used in a bi-linear filtering operation.

Syntax

Syntax

```
TemplateType GatherGreen(  
    in SamplerState S,  
    in float2      Location,  
    in int2       Offset1,  
    in int2       Offset2,  
    in int2       Offset3,  
    in int2       Offset4  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

Offset1 [in]

Type: **int2**

The first offset component applied to the texture coordinates before sampling.

Offset2 [in]

Type: **int2**

The second offset component applied to the texture coordinates before sampling.

Offset3 [in]

Type: `int2`

The third offset component applied to the texture coordinates before sampling.

Offset4 [in]

Type: `int2`

The fourth offset component applied to the texture coordinates before sampling.

Return value

Type: `TemplateType`

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherGreen methods](#)

Texture2D::GatherGreen(S,float,int,uint) function

Article • 03/06/2023 • 2 minutes to read

Returns the green components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

Syntax

Syntax

```
TemplateType GatherGreen(  
    in  SamplerState S,  
    in  float2      Location,  
    in  int         Offset,  
    out uint       Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

Offset [in]

Type: **int**

The offset applied to the texture coordinates before sampling.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns **FALSE**.

Return value

Type: [TemplateType](#)

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherGreen methods](#)

Texture2D::GatherGreen(S,float,int2,int2,int2,int2,uint) function

Article • 03/06/2023 • 2 minutes to read

Returns the green components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

Syntax

Syntax

```
TemplateType GatherGreen(  
    in SamplerState S,  
    in float2      Location,  
    in int2        Offset1,  
    in int2        Offset2,  
    in int2        Offset3,  
    in int2        Offset4,  
    out uint       Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

Offset1 [in]

Type: **int2**

The first offset component applied to the texture coordinates before sampling.

Offset2 [in]

Type: **int2**

The second offset component applied to the texture coordinates before sampling.

Offset3 [in]

Type: **int2**

The third offset component applied to the texture coordinates before sampling.

Offset4 [in]

Type: **int2**

The fourth offset component applied to the texture coordinates before sampling.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. **CheckAccessFullyMapped** returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, **CheckAccessFullyMapped** returns **FALSE**.

Return value

Type: **TemplateType**

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherGreen methods](#)

Texture2D::GatherRed methods

Article • 02/12/2021 • 2 minutes to read

Returns the red components of a [Texture2D](#)'s four texel values that would be used in a bi-linear filtering operation.

Overload list

Method	Description
GatherRed(S,float,int)	Returns the red components of the four texel values that would be used in a bi-linear filtering operation.
GatherRed(S,float,int,uint)	Returns the red components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.
GatherRed(S,float,int2,int2,int2,int2)	Returns the red components of the four texel values that would be used in a bi-linear filtering operation.
GatherRed(S,float,int2,int2,int2,int2,uint)	Returns the red components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

See also

[Texture2D](#)

Texture2D::GatherRed(S,float,int) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their red component against a compare value.

Syntax

Syntax

```
TemplateType GatherRed(  
    in sampler s,  
    in float2 location,  
    in int2 offset  
) ;
```

Parameters

s [in]

Type: **sampler**

The zero-based sampler index.

location [in]

Type: **float2**

The sample coordinates (u,v).

offset [in]

Type: **int2**

An offset that is applied to the texture coordinate before sampling.

Return value

Type: **TemplateType**

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherRed methods](#)

[Shader Model 5](#)

Texture2D::GatherRed(S,float,int2,int2,int2,int2) function

Article • 03/06/2023 • 2 minutes to read

Returns the red components of the four texel values that would be used in a bi-linear filtering operation.

Syntax

Syntax

```
TemplateType GatherRed(  
    in SamplerState S,  
    in float2      Location,  
    in int2       Offset1,  
    in int2       Offset2,  
    in int2       Offset3,  
    in int2       Offset4  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

Offset1 [in]

Type: **int2**

The first offset component applied to the texture coordinates before sampling.

Offset2 [in]

Type: **int2**

The second offset component applied to the texture coordinates before sampling.

Offset3 [in]

Type: `int2`

The third offset component applied to the texture coordinates before sampling.

Offset4 [in]

Type: `int2`

The fourth offset component applied to the texture coordinates before sampling.

Return value

Type: `TemplateType`

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherRed methods](#)

Texture2D::GatherRed(S,float,int,uint) function

Article • 03/06/2023 • 2 minutes to read

Returns the red components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

Syntax

Syntax

```
TemplateType GatherRed(  
    in  SamplerState S,  
    in  float2      Location,  
    in  int         Offset,  
    out uint       Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

Offset [in]

Type: **int**

The offset applied to the texture coordinates before sampling.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns **FALSE**.

Return value

Type: [TemplateType](#)

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherRed methods](#)

Texture2D::GatherRed(S,float,int2,int2,int2,int2,uint) function

Article • 03/06/2023 • 2 minutes to read

Returns the red components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

Syntax

Syntax

```
TemplateType GatherRed(  
    in SamplerState S,  
    in float2      Location,  
    in int2       Offset1,  
    in int2       Offset2,  
    in int2       Offset3,  
    in int2       Offset4,  
    out uint      Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

Offset1 [in]

Type: **int2**

The first offset component applied to the texture coordinates before sampling.

Offset2 [in]

Type: **int2**

The second offset component applied to the texture coordinates before sampling.

Offset3 [in]

Type: **int2**

The third offset component applied to the texture coordinates before sampling.

Offset4 [in]

Type: **int2**

The fourth offset component applied to the texture coordinates before sampling.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. **CheckAccessFullyMapped** returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, **CheckAccessFullyMapped** returns **FALSE**.

Return value

Type: **TemplateType**

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherRed methods](#)

Texture2D::GetDimensions function

Article • 03/09/2021 • 2 minutes to read

Returns the dimensions of the resource.

Syntax

```
syntax

void GetDimensions(
    in
        uint MipLevel,
    out
        uint Width,
    out uint Height,
    out
        uint NumberOfLevels
);
```

Parameters

MipLevel [in]

Type: **uint**

Optional. The mipmap level (must be specified if *NumberOfLevels* is used).

Width [out]

Type: **uint**

The resource width, in texels.

Height [out]

Type: **uint**

The resource height, in texels.

NumberOfLevels [out]

Type: **uint**

The number of mipmap levels (requires *MipLevel* also).

Return value

Nothing

Remarks

This is a list of the overloaded versions of this method.

```
void GetDimensions(uint MipLevel,  
    out uint Width,  
    out uint Height,  
    out uint NumberOfLevels);  
  
void GetDimensions (out uint Width,  
    out uint Height);  
  
void GetDimensions(uint MipLevel,  
    out float Width,  
    out float Height,  
    out float NumberOfLevels);  
  
void GetDimensions(out float Width,  
    out float Height);
```

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	x

See also

[Texture2D](#)

[Shader Model 5](#)

Texture2D::Load methods

Article • 11/06/2019 • 2 minutes to read

Reads texture data from a [Texture2D](#).

Overload list

Method	Description
Load(int,int)	Reads texture data.
Load(int,int,uint)	Reads texture data and returns status of the operation.

See also

[Texture2D](#)

Texture2D::Load(int,int,uint) function

Article • 03/09/2021 • 2 minutes to read

Reads texture data and returns status of the operation.

Syntax

syntax

```
Load(  
    in int Location,  
    in int Offset,  
    out uint Status  
) ;
```

Parameters

Location [in]

Type: **int**

The texture coordinates.

Offset [in]

Type: **int**

An offset applied to the texture coordinates before sampling.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns **TRUE** if all values from the corresponding `Sample`, `Gather`, or `Load` operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns **FALSE**.

Return value

Type:

The return type matches the type in the declaration for the [Texture2D](#) object.

See also

[Load methods](#)

Texture2D::mips.Operator function

Article • 03/09/2021 • 2 minutes to read

Returns a read-only resource variable.

Syntax

syntax

```
R mips.Operator[][](  
    in uint mipSlice,  
    in uint2 pos  
)
```

Parameters

mipSlice [in]

Type: **uint**

The mip slice index.

pos [in]

Type: **uint2**

The index position. Contains the (x, y) coordinates.

Return value

Type: **R**

A read-only resource variable.

Remarks

Usage Example

```
Texture2D<float4> tex;  
uint mip = 2;  
uint2 pos_xy = {123, 456};  
float4 f = tex.mips[mip][pos_xy];
```

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

See also

[Texture2D](#)

[Shader Model 5](#)

Texture2D::Operator function

Article • 03/09/2021 • 2 minutes to read

Returns a read-only resource variable.

Syntax

syntax

```
R Operator[](  
    in uint2 pos  
) ;
```

Parameters

pos [in]

Type: **uint2**

The index position. Contains the (x, y) coordinates.

Return value

Type: **R**

A read-only resource variable.

Remarks

This method always accesses the first mip level. To specify other mip levels, use the [mip.operator\[\]\[\]](#) method instead.

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Texture2D](#)

[Shader Model 5](#)

Texture2D::Sample methods

Article • 10/24/2019 • 2 minutes to read

Samples a [Texture2D](#).

Overload list

Method	Description
Sample(S,float,int)	Samples a texture.
Sample(S,float,int,float)	Samples a texture with an optional value to clamp sample level-of-detail (LOD) values to.
Sample(S,float,int,float,uint)	Samples a texture with an optional value to clamp sample level-of-detail (LOD) values to, and returns status of the operation.

Remarks

Refer to [Texture-Object](#).

See also

[Texture2D](#)

Sample(S,float,int,float) function (HLSL reference)

Article • 11/17/2020 • 2 minutes to read

Samples a [Texture2D](#) with an optional value to clamp sample level-of-detail (LOD) values to.

ⓘ Note

Requires shader model 5 or higher.

Syntax

syntax

```
DXGI_FORMAT Sample(  
    in SamplerState S,  
    in float Location,  
    in int Offset,  
    in float Clamp  
) ;
```

Parameters

S [in]

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2

Texture-Object Type	Parameter Type
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

Offset [in]

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. The texture offsets need to be static. The argument type is dependent on the texture-object type. For more info, see [Applying texture coordinate offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Return value

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

Remarks

Texture sampling uses the texel position to look up a texel value. An offset can be applied to the position before lookup. The sampler state contains the sampling and filtering options. This method can be invoked within a pixel shader, but it is not supported in a vertex shader or a geometry shader.

Use an offset only at an integer miplevel; otherwise, you may get different results depending on hardware implementation or driver settings.

Calculating Texel Positions

Texture coordinates are floating-point values that reference texture data, which is also known as normalized texture space. Address wrapping modes are applied in this order (texture coordinates + offsets + wrap mode) to modify texture coordinates outside the [0..1] range.

For texture arrays, an additional value in the location parameter specifies an index into a texture array. This index is treated as a scaled float value (instead of the normalized space for standard texture coordinates). The conversion to an integer index is done in the following order (float + round-to-nearest-even integer + clamp to the array range).

Applying Texture Coordinate Offsets

The offset parameter modifies the texture coordinates, in texel space. Even though texture coordinates are normalized floating-point numbers, the offset applies an integer offset. Also note that the texture offsets need to be static.

The data format returned is determined by the texture format. For example, if the texture resource was defined with the DXGI_FORMAT_A8B8G8R8_UNORM_SRGB format, the sampling operation converts sampled texels from gamma 2.0 to 1.0, filter, and writes the result as a floating-point value in the range [0..1].

Use an offset only at an integer mipmap; otherwise, you may get results that do not translate well to hardware.

See also

[Sample methods](#)

[Texture-Object](#)

Sample(S,float,int,float,uint) function (HLSL reference)

Article • 11/17/2020 • 2 minutes to read

Samples a [Texture2D](#) with an optional value to clamp sample level-of-detail (LOD) values to, and returns status of the operation.

ⓘ Note

Requires shader model 5 or higher.

Syntax

```
syntax

DXGI_FORMAT Sample(
    in SamplerState S,
    in float Location,
    in int Offset,
    in float Clamp,
    out uint Status
);
```

Parameters

S [in]

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2

Texture-Object Type	Parameter Type
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

Offset [in]

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. The texture offsets need to be static. The argument type is dependent on the texture-object type. For more info, see [Applying texture coordinate offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Status [out]

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns **FALSE**.

Return value

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

Remarks

Texture sampling uses the texel position to look up a texel value. An offset can be applied to the position before lookup. The sampler state contains the sampling and filtering options. This method can be invoked within a pixel shader, but it is not supported in a vertex shader or a geometry shader.

Use an offset only at an integer mipmap; otherwise, you may get different results depending on hardware implementation or driver settings.

Calculating Texel Positions

Texture coordinates are floating-point values that reference texture data, which is also known as normalized texture space. Address wrapping modes are applied in this order (texture coordinates + offsets + wrap mode) to modify texture coordinates outside the [0...1] range.

For texture arrays, an additional value in the location parameter specifies an index into a texture array. This index is treated as a scaled float value (instead of the normalized space for standard texture coordinates). The conversion to an integer index is done in the following order (float + round-to-nearest-even integer + clamp to the array range).

Applying Texture Coordinate Offsets

The offset parameter modifies the texture coordinates, in texel space. Even though texture coordinates are normalized floating-point numbers, the offset applies an integer offset. Also note that the texture offsets need to be static.

The data format returned is determined by the texture format. For example, if the texture resource was defined with the DXGI_FORMAT_A8B8G8R8_UNORM_SRGB format, the sampling operation converts sampled texels from gamma 2.0 to 1.0, filter, and writes the result as a floating-point value in the range [0..1].

See also

[Sample methods](#)

[Texture-Object](#)

Texture2D::SampleBias methods

Article • 11/06/2019 • 2 minutes to read

Samples a [Texture2D](#), after applying the bias value to the mipmap level.

Overload list

Method	Description
SampleBias(S,float,float,int)	Samples a texture, after applying the bias value to the mipmap level.
SampleBias(S,float,float,int,float)	Samples a texture, after applying the bias value to the mipmap level, with an optional value to clamp sample level-of-detail (LOD) values to.
SampleBias(S,float,float,int,float,uint)	Samples a texture, after applying the bias value to the mipmap level, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

Remarks

Refer to [Texture-Object](#).

See also

[Texture2D](#)

SampleBias::SampleBias(S,float,float,int,float) function for Texture2D

Article • 04/02/2021 • 2 minutes to read

Samples a [Texture2D](#), after applying the bias value to the mipmap level, with an optional value to clamp sample level-of-detail (LOD) values to.

Syntax

Syntax

```
DXGI_FORMAT SampleBias(  
    in SamplerState S,  
    in float        Location,  
    in float        Bias,  
    in int         Offset,  
    in float        Clamp  
) ;
```

Parameters

S [in]

Type: [SamplerState](#)

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: [float](#)

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

Bias [in]

Type: **float**

The bias value, which is a floating-point number between 0.0 and 1.0 inclusive, is applied to a mip level before sampling.

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer miplevel; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

SampleBias methods

SampleBias::SampleBias(S,float,float,int,float,uint) function for Texture2D

Article • 03/15/2021 • 2 minutes to read

Samples a [Texture2D](#), after applying the bias value to the mipmap level, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

Syntax

syntax

```
DXGI_FORMAT SampleBias(  
    in SamplerState S,  
    in float        Location,  
    in float        Bias,  
    in int         Offset,  
    in float        Clamp,  
    out uint       Status  
) ;
```

Parameters

S [in]

Type: [SamplerState](#)

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: [float](#)

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3

Texture-Object Type	Parameter Type
TextureCubeArray	float4

Bias [in]

Type: **float**

The bias value, which is a floating-point number between 0.0 and 1.0 inclusive, is applied to a mip level before sampling.

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer miplevel; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns

TRUE if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, **CheckAccessFullyMapped** returns FALSE.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[SampleBias methods](#)

Texture2D::SampleCmp methods

Article • 11/06/2019 • 2 minutes to read

Samples a [Texture2D](#), using a comparison value to reject samples.

Overload list

Method	Description
SampleCmp(S,float,float,int)	Samples a texture, using a comparison value to reject samples.
SampleCmp(S,float,float,int,float)	Samples a texture, using a comparison value to reject samples, with an optional value to clamp sample level-of-detail (LOD) values to.
SampleCmp(S,float,float,int,float,uint)	Samples a texture, using a comparison value to reject samples, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

Remarks

Refer to [Texture-Object](#).

See also

[Texture2D](#)

SampleCmp::SampleCmp(S,float,float,int,float) function for Texture2D

Article • 03/15/2021 • 2 minutes to read

Samples a [Texture2D](#), using a comparison value to reject samples, with an optional value to clamp sample level-of-detail (LOD) values to.

Syntax

Syntax

```
DXGI_FORMAT SampleCmp(  
    in SamplerState S,  
    in float        Location,  
    in float        CompareValue,  
    in int          Offset,  
    in float        Clamp  
) ;
```

Parameters

S [in]

Type: [SamplerState](#)

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: [float](#)

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

CompareValue [in]

Type: **float**

A floating-point value to use as a comparison value.

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer miplevel; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[SampleCmp methods](#)

SampleCmp::SampleCmp(S,float,float,int,float,uint) function for Texture2D

Article • 03/15/2021 • 2 minutes to read

Samples a [Texture2D](#), using a comparison value to reject samples, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

Syntax

```
syntax

DXGI_FORMAT SampleCmp(
    in SamplerState S,
    in float        Location,
    in float        CompareValue,
    in int         Offset,
    in float        Clamp,
    out uint       Status
);
```

Parameters

S [in]

Type: [SamplerState](#)

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: [float](#)

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3

Texture-Object Type	Parameter Type
TextureCubeArray	float4

CompareValue [in]

Type: **float**

A floating-point value to use as a comparison value.

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer mipmap level; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns TRUE if all values from the corresponding [Sample](#), [Gather](#), or [Load](#) operation accessed

mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns `FALSE`.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[SampleCmp methods](#)

Texture2D::SampleCmpLevelZero methods

Article • 11/06/2019 • 2 minutes to read

Samples a [Texture2D](#) on mipmap level 0 only, using a comparison value to reject samples.

Overload list

Method	Description
SampleCmpLevelZero(S,float,float,int)	Samples a texture on mipmap level 0 only and compares the result to a comparison value.
SampleCmpLevelZero(S,float,float,int,uint)	Samples a texture on mipmap level 0 only and compares the result to a comparison value. Returns status about the operation.

Remarks

Refer to [Texture-Object](#).

See also

[Texture2D](#)

SampleCmpLevelZero::SampleCmpLevelZero(S,float,float,int,uint) function for Texture2D

Article • 03/15/2021 • 2 minutes to read

Samples a [Texture2D](#) on mipmap level 0 only and compares the result to a comparison value. Returns status about the operation.

Syntax

```
syntax

DXGI_FORMAT SampleCmpLevelZero(
    in SamplerState S,
    in float          Location,
    in float          CompareValue,
    in int           Offset,
    out uint         Status
);
```

Parameters

S [in]

Type: [SamplerState](#)

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: [float](#)

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3

Texture-Object Type	Parameter Type
TextureCubeArray	float4

CompareValue [in]

Type: **float**

A floating-point value to use as a comparison value.

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer mipmap level; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns **FALSE**.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[SampleCmpLevelZero methods](#)

Texture2D::SampleGrad methods

Article • 11/06/2019 • 2 minutes to read

Samples a [Texture2D](#) using a gradient to influence the way the sample location is calculated.

Overload list

Method	Description
SampleGrad(S,float,float,float,int)	Samples a texture, using a gradient to influence the way the sample location is calculated.
SampleGrad(S,float,float,float,int,float)	Samples a texture, using a gradient to influence the way the sample location is calculated, with an optional value to clamp sample level-of-detail (LOD) values to.
SampleGrad(S,float,float,float,int,float,uint)	Samples a texture, using a gradient to influence the way the sample location is calculated, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

Remarks

Refer to [Texture-Object](#).

See also

[Texture2D](#)

SampleGrad::SampleGrad(S,float,float, float,int,float) function for Texture2D

Article • 03/15/2021 • 2 minutes to read

Samples a [Texture2D](#), using a gradient to influence the way the sample location is calculated, with an optional value to clamp sample level-of-detail (LOD) values to.

Syntax

Syntax

```
DXGI_FORMAT SampleGrad(  
    in SamplerState S,  
    in float        Location,  
    in float        DDX,  
    in float        DDY,  
    in int          Offset,  
    in float        Clamp  
) ;
```

Parameters

S [in]

Type: [SamplerState](#)

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: [float](#)

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3

Texture-Object Type	Parameter Type
TextureCubeArray	float4

DDX [in]

Type: **float**

The rate of change of the surface geometry in the x direction. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	float
Texture2D, Texture2DArray	float2
Texture3D, TextureCube, TextureCubeArray	float3
Texture2DMS, Texture2DMSArray	not supported

DDY [in]

Type: **float**

The rate of change of the surface geometry in the y direction. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	float
Texture2D, Texture2DArray	float2
Texture3D, TextureCube, TextureCubeArray	float3
Texture2DMS, Texture2DMSArray	not supported

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer miplevel; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Return value

Type: **DXGI_FORMAT**

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[SampleGrad methods](#)

SampleGrad::SampleGrad(S,float,float,fl oat,int,float,uint) function for Texture2D

Article • 03/15/2021 • 2 minutes to read

Samples a [Texture2D](#), using a gradient to influence the way the sample location is calculated, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

Syntax

syntax

```
DXGI_FORMAT SampleGrad(  
    in SamplerState S,  
    in float        Location,  
    in float        DDX,  
    in float        DDY,  
    in int          Offset,  
    in float        Clamp,  
    out uint        Status  
) ;
```

Parameters

S [in]

Type: [SamplerState](#)

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: [float](#)

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2

Texture-Object Type	Parameter Type
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

DDX [in]

Type: **float**

The rate of change of the surface geometry in the x direction. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	float
Texture2D, Texture2DArray	float2
Texture3D, TextureCube, TextureCubeArray	float3
Texture2DMS, Texture2DMSArray	not supported

DDY [in]

Type: **float**

The rate of change of the surface geometry in the y direction. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	float
Texture2D, Texture2DArray	float2
Texture3D, TextureCube, TextureCubeArray	float3
Texture2DMS, Texture2DMSArray	not supported

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer miplevel; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns **TRUE** if all values from the corresponding [Sample](#), [Gather](#), or [Load](#) operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns **FALSE**.

Return value

Type: [**DXGI_FORMAT**](#)

The texture format, which is one of the typed values listed in [**DXGI_FORMAT**](#).

See also

[SampleGrad methods](#)

Texture2D::SampleLevel methods

Article • 11/06/2019 • 2 minutes to read

Samples a [Texture2D](#) on the specified mipmap level.

Overload list

Method	Description
SampleLevel(S,float,float,int)	Samples a texture on the specified mipmap level.
SampleLevel(S,float,float,int,uint)	Samples a texture on the specified mipmap level and returns status about the operation.

Remarks

Refer to [Texture-Object](#).

See also

[Texture2D](#)

SampleLevel::SampleLevel(S,float,float,int,uint) function for Texture2D

Article • 03/15/2021 • 2 minutes to read

Samples a [Texture2D](#) on the specified mipmap level and returns status about the operation.

Syntax

Syntax

```
DXGI_FORMAT SampleLevel(  
    in SamplerState S,  
    in float        Location,  
    in float        LOD,  
    in int         Offset,  
    out uint       Status  
) ;
```

Parameters

S [in]

Type: [SamplerState](#)

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: [float](#)

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

LOD [in]

Type: **float**

[in] A number that specifies the mipmap level. If the value is ≤ 0 , mipmap level 0 (biggest map) is used. The fractional value (if supplied) is used to interpolate between two mipmap levels.

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer miplevel; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns **FALSE**.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[SampleLevel methods](#)

Texture2DArray

Article • 02/12/2021 • 2 minutes to read

Texture2DArray type ([as it exists in Shader Model 4](#)) plus resource variables. This texture object supports the following methods in addition to the methods in Shader Model 4.

Method	Description
Gather	Returns the four texel values that would be used in a bi-linear filtering operation.
GatherRed	Returns the red components of the four texel values that would be used in a bi-linear filtering operation.
GatherGreen	Returns the green components of the four texel values that would be used in a bi-linear filtering operation.
GatherBlue	Returns the blue components of the four texel values that would be used in a bi-linear filtering operation.
GatherAlpha	Returns the alpha components of the four texel values that would be used in a bi-linear filtering operation.
GatherCmp	For four texel values that would be used in a bi-linear filtering operation, returns their comparison against a compare value.
GatherCmpRed	For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their red component against a compare value.
GatherCmpGreen	For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their green component against a compare value.
GatherCmpBlue	For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their blue component against a compare value.
GatherCmpAlpha	For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their alpha component against a compare value.
GetDimensions	Gets the resource dimensions.
Load	Reads texture data.
mips.Operator[][]	Gets a read-only resource variable.
Operator[]	Gets a read-only resource variable.
Sample	Samples a texture.
SampleBias	Samples a texture, after applying the bias value to the mipmap level.

Method	Description
SampleCmp	Samples a texture, using a comparison value to reject samples.
SampleCmpLevelZero	Samples a texture (mipmap level 0 only), using a comparison value to reject samples.
SampleGrad	Samples a texture using a gradient to influence the way the sample location is calculated.
SampleLevel	Samples a texture on the specified mipmap level.

Minimum Shader Model

This object is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This object is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Shader Model 5 Objects](#)

Texture2DArray::Gather methods

Article • 03/05/2021 • 2 minutes to read

Returns the four texel values of a [Texture2DArray](#) that would be used in a bi-linear filtering operation.

See the documentation on [gather4](#) for more information describing the underlying DXBC instruction.

Overload list

Method	Description
Gather(S,float,int)	Returns the four texel values that would be used in a bi-linear filtering operation.
Gather(S,float,int,uint)	Returns the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

See also

[Texture2DArray](#)

Texture2DArray::Gather(S,float,int) function

Article • 03/09/2021 • 2 minutes to read

Returns the four texel values that would be used in a bi-linear filtering operation.

Syntax

syntax

```
TemplateType Gather(  
    in sampler s,  
    in float3 location,  
    in int2 offset  
) ;
```

Parameters

s [in]

Type: **sampler**

The zero-based sampler index.

location [in]

Type: **float3**

The sample coordinates (u,v).

offset [in]

Type: **int2**

The offset applied to the texture coordinates before sampling.

Return value

Type: **TemplateType**

A four-component value whose type is the same as the template type.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Gather methods](#)

[Shader Model 5](#)

Texture2DArray::Gather(S,float,int,uint) function

Article • 03/06/2023 • 2 minutes to read

Returns the four texel values that would be used in a bi-linear filtering operation.

Syntax

syntax

```
TemplateType Gather(  
    in SamplerState S,  
    in float3      Location,  
    in int         Offset,  
    out uint       Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

Offset [in]

Type: **int**

The offset applied to the texture coordinates before sampling.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns

TRUE if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, **CheckAccessFullyMapped** returns FALSE.

Return value

Type: **TemplateType**

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

See also

[Gather methods](#)

Texture2DArray::GatherAlpha methods

Article • 02/12/2021 • 2 minutes to read

Returns the alpha components of a [Texture2DArray](#)'s four texel values that would be used in a bi-linear filtering operation.

Overload list

Method	Description
GatherAlpha(S,float,int)	Returns the alpha components of the four texel values that would be used in a bi-linear filtering operation.
GatherAlpha(S,float,int,uint)	Returns the alpha components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.
GatherAlpha(S,float,int2,int2,int2,int2)	Returns the alpha components of the four texel values that would be used in a bi-linear filtering operation.
GatherAlpha(S,float,int2,int2,int2,int2,uint)	Returns the alpha components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

See also

[Texture2DArray](#)

Texture2DArray::GatherAlpha(S,float,int) function

Article • 03/09/2021 • 2 minutes to read

Returns the alpha components of the four texel values that would be used in a bi-linear filtering operation.

Syntax

Syntax

```
TemplateType GatherAlpha(  
    in sampler s,  
    in float3 location,  
    in int2 offset  
) ;
```

Parameters

s [in]

Type: **sampler**

The zero-based sampler index.

location [in]

Type: **float3**

The sample coordinates (u,v).

offset [in]

Type: **int2**

An offset that is applied to the texture coordinate before sampling.

Return value

Type: **TemplateType**

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherAlpha methods](#)

[Shader Model 5](#)

Texture2DArray::GatherAlpha(S,float,int2,int2,int2) function

Article • 03/06/2023 • 2 minutes to read

Returns the alpha components of the four texel values that would be used in a bi-linear filtering operation.

Syntax

Syntax

```
TemplateType GatherAlpha(  
    in SamplerState S,  
    in float3      Location,  
    in int2        Offset1,  
    in int2        Offset2,  
    in int2        Offset3,  
    in int2        Offset4  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

Offset1 [in]

Type: **int2**

The first offset component applied to the texture coordinates before sampling.

Offset2 [in]

Type: **int2**

The second offset component applied to the texture coordinates before sampling.

Offset3 [in]

Type: **int2**

The third offset component applied to the texture coordinates before sampling.

Offset4 [in]

Type: **int2**

The fourth offset component applied to the texture coordinates before sampling.

Return value

Type: **TemplateType**

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherAlpha methods](#)

Texture2DArray::GatherAlpha(S,float,int,uint) function

Article • 03/06/2023 • 2 minutes to read

Returns the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

Syntax

Syntax

```
TemplateType GatherAlpha(  
    in  SamplerState S,  
    in  float3      Location,  
    in  int         Offset,  
    out uint        Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

Offset [in]

Type: **int**

The offset applied to the texture coordinates before sampling.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns **FALSE**.

Return value

Type: [TemplateType](#)

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherAlpha methods](#)

Texture2DArray::GatherAlpha(S,float,int2,int2,int2,uint) function

Article • 03/06/2023 • 2 minutes to read

Returns the alpha components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

Syntax

Syntax

```
TemplateType GatherAlpha(  
    in SamplerState S,  
    in float3      Location,  
    in int2        Offset1,  
    in int2        Offset2,  
    in int2        Offset3,  
    in int2        Offset4,  
    out uint       Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

Offset1 [in]

Type: **int2**

The first offset component applied to the texture coordinates before sampling.

Offset2 [in]

Type: **int2**

The second offset component applied to the texture coordinates before sampling.

Offset3 [in]

Type: **int2**

The third offset component applied to the texture coordinates before sampling.

Offset4 [in]

Type: **int2**

The fourth offset component applied to the texture coordinates before sampling.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. **CheckAccessFullyMapped** returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, **CheckAccessFullyMapped** returns **FALSE**.

Return value

Type: **TemplateType**

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherAlpha methods](#)

Texture2DArray::GatherBlue methods

Article • 02/12/2021 • 2 minutes to read

Returns the blue components of a [Texture2DArray](#)'s four texel values that would be used in a bi-linear filtering operation.

Overload list

Method	Description
GatherBlue(S,float,int)	Returns the blue components of the four texel values that would be used in a bi-linear filtering operation.
GatherBlue(S,float,int,uint)	Returns the blue components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.
GatherBlue(S,float,int2,int2,int2,int2)	Returns the blue components of the four texel values that would be used in a bi-linear filtering operation.
GatherBlue(S,float,int2,int2,int2,int2,uint)	Returns the blue components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

See also

[Texture2DArray](#)

Texture2DArray::GatherBlue(S,float,int) function

Article • 03/09/2021 • 2 minutes to read

Returns the blue components of the four texel values that would be used in a bi-linear filtering operation.

Syntax

Syntax

```
TemplateType GatherBlue(  
    in sampler s,  
    in float3 location,  
    in int2 offset  
) ;
```

Parameters

s [in]

Type: **sampler**

The zero-based sampler index.

location [in]

Type: **float3**

The sample coordinates (u,v).

offset [in]

Type: **int2**

An offset that is applied to the texture coordinate before sampling.

Return value

Type: **TemplateType**

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherBlue methods](#)

[Shader Model 5](#)

GatherBlue(S,float,int2,int2,int2,int2)

function (HLSL reference)

Article • 03/06/2023 • 2 minutes to read

Returns the blue components of the four texel values that would be used in a bi-linear filtering operation.

Syntax

Syntax

```
TemplateType GatherBlue(  
    in SamplerState S,  
    in float3      Location,  
    in int2        Offset1,  
    in int2        Offset2,  
    in int2        Offset3,  
    in int2        Offset4  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

Offset1 [in]

Type: **int2**

The first offset component applied to the texture coordinates before sampling.

Offset2 [in]

Type: **int2**

The second offset component applied to the texture coordinates before sampling.

Offset3 [in]

Type: `int2`

The third offset component applied to the texture coordinates before sampling.

Offset4 [in]

Type: `int2`

The fourth offset component applied to the texture coordinates before sampling.

Return value

Type: `TemplateType`

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherBlue methods](#)

Texture2DArray::GatherBlue(S,float,int,uint) function

Article • 03/06/2023 • 2 minutes to read

Returns the blue components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

Syntax

Syntax

```
TemplateType GatherBlue(  
    in  SamplerState S,  
    in  float3      Location,  
    in  int         Offset,  
    out uint       Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

Offset [in]

Type: **int**

The offset applied to the texture coordinates before sampling.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns **FALSE**.

Return value

Type: [TemplateType](#)

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherBlue methods](#)

Texture2DArray::GatherBlue(S,float,int2,int2,int2,uint) function

Article • 03/06/2023 • 2 minutes to read

Returns the blue components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

Syntax

Syntax

```
TemplateType GatherBlue(  
    in SamplerState S,  
    in float3      Location,  
    in int2        Offset1,  
    in int2        Offset2,  
    in int2        Offset3,  
    in int2        Offset4,  
    out uint       Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

Offset1 [in]

Type: **int2**

The first offset component applied to the texture coordinates before sampling.

Offset2 [in]

Type: **int2**

The second offset component applied to the texture coordinates before sampling.

Offset3 [in]

Type: `int2`

The third offset component applied to the texture coordinates before sampling.

Offset4 [in]

Type: `int2`

The fourth offset component applied to the texture coordinates before sampling.

Status [out]

Type: `uint`

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns `TRUE` if all values from the corresponding `Sample`, `Gather`, or `Load` operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns `FALSE`.

Return value

Type: `TemplateType`

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherBlue methods](#)

Texture2DArray::GatherCmp methods

Article • 03/05/2021 • 2 minutes to read

For four texel values of a [Texture2DArray](#) that would be used in a bi-linear filtering operation, returns their comparison against a compare value.

See the documentation on [gather4_c](#) for more information describing the underlying DXBC instruction.

Overload list

Method	Description
GatherCmp(S,float,float,int)	Samples and compares a texture and returns all four components.
GatherCmp(S,float,float,int,uint)	Samples and compares a texture and returns all four components along with status about the operation.

See also

[Texture2DArray](#)

Texture2DArray::GatherCmp(S,float,float,int) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns their comparison against a compare value.

Syntax

Syntax

```
float4 GatherCmp(  
    in SamplerComparisonState s,  
    in float3 location,  
    in float compare_value,  
    in int2 offset  
) ;
```

Parameters

s [in]

Type: **SamplerComparisonState**

The zero-based sampler index.

location [in]

Type: **float3**

The sample coordinates (u,v).

compare_value [in]

Type: **float**

A value to compare each against each sampled value.

offset [in]

Type: **int2**

An offset that is applied to the texture coordinate before sampling.

Return value

Type: **float4**

A four-component value, each component is the result of a per-component comparison.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmp methods](#)

[Shader Model 5](#)

Texture2DArray::GatherCmp(S,float,float,int,uint) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns their comparison against a compare value along with tile-mapping status.

Syntax

Syntax

```
TemplateType GatherCmp(  
    in SamplerState S,  
    in float        Location,  
    in float        CompareValue,  
    in int          Offset,  
    out uint        Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

CompareValue [in]

Type: **float**

A value to compare each against each sampled value.

Offset [in]

Type: **int**

The offset applied to the texture coordinates before sampling.

Status [out]

Type: `uint`

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns `TRUE` if all values from the corresponding `Sample`, `Gather`, or `Load` operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns `FALSE`.

Return value

Type: `TemplateType`

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmp methods](#)

Texture2DArray::GatherCmpAlpha methods

Article • 11/06/2019 • 2 minutes to read

Samples and compares a [Texture2DArray](#) and returns the alpha component.

Overload list

Method	Description
GatherCmpAlpha(S,float,float,int)	Samples and compares a texture and returns the alpha component.
GatherCmpAlpha(S,float,float,int,uint)	Samples and compares a texture and returns the alpha component along with status about the operation.
GatherCmpAlpha(S,float,float,int2,int2,int2,int2)	Samples and compares a texture and returns the alpha component.
GatherCmpAlpha(S,float,float,int2,int2,int2,int2,uint)	Samples and compares a texture and returns the alpha component along with status about the operation.

See also

[Texture2DArray](#)

Texture2DArray::GatherCmpAlpha(S,float,int) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their alpha component against a compare value.

Syntax

Syntax

```
float4 GatherCmpAlpha(  
    in SamplerComparisonState s,  
    in float3 location,  
    in float compare_value,  
    in int2 offset  
) ;
```

Parameters

s [in]

Type: **SamplerComparisonState**

The zero-based sampler index.

location [in]

Type: **float3**

The sample coordinates (u,v).

compare_value [in]

Type: **float**

A value to compare each against each sampled value.

offset [in]

Type: **int2**

An offset that is applied to the texture coordinate before sampling.

Return value

Type: **float4**

A four-component value, each component is the result of a per-component comparison.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmpAlpha methods](#)

[Shader Model 5](#)

Texture2DArray::GatherCmpAlpha(S,float,int2,int2,int2,int2) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their alpha component against a compare value.

Syntax

Syntax

```
TemplateType GatherCmpAlpha(  
    in SamplerState S,  
    in float      Location,  
    in float      CompareValue,  
    in int2       Offset1,  
    in int2       Offset2,  
    in int2       Offset3,  
    in int2       Offset4  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

CompareValue [in]

Type: **float**

A value to compare each against each sampled value.

Offset1 [in]

Type: **int2**

The first offset component applied to the texture coordinates before sampling.

Offset2 [in]

Type: **int2**

The second offset component applied to the texture coordinates before sampling.

Offset3 [in]

Type: **int2**

The third offset component applied to the texture coordinates before sampling.

Offset4 [in]

Type: **int2**

The fourth offset component applied to the texture coordinates before sampling.

Return value

Type: **TemplateType**

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmpAlpha methods](#)

Texture2DArray::GatherCmpAlpha(S,float,int,uint) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their alpha component against a compare value along with tile-mapping status.

Syntax

syntax

```
TemplateType GatherCmpAlpha(  
    in SamplerState S,  
    in float      Location,  
    in float      CompareValue,  
    in int       Offset,  
    out uint     Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

CompareValue [in]

Type: **float**

A value to compare each against each sampled value.

Offset [in]

Type: **int**

The offset applied to the texture coordinates before sampling.

Status [out]

Type: `uint`

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns `TRUE` if all values from the corresponding `Sample`, `Gather`, or `Load` operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns `FALSE`.

Return value

Type: `TemplateType`

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmpAlpha methods](#)

Texture2DArray::GatherCmpAlpha(S,float,int2,int2,int2,int2,uint) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their alpha component against a compare value along with tile-mapping status.

Syntax

syntax

```
TemplateType GatherCmpAlpha(  
    in SamplerState S,  
    in float      Location,  
    in float      CompareValue,  
    in int2       Offset1,  
    in int2       Offset2,  
    in int2       Offset3,  
    in int2       Offset4,  
    out uint     Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

CompareValue [in]

Type: **float**

A value to compare each against each sampled value.

Offset1 [in]

Type: **int2**

The first offset component applied to the texture coordinates before sampling.

Offset2 [in]

Type: **int2**

The second offset component applied to the texture coordinates before sampling.

Offset3 [in]

Type: **int2**

The third offset component applied to the texture coordinates before sampling.

Offset4 [in]

Type: **int2**

The fourth offset component applied to the texture coordinates before sampling.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. **CheckAccessFullyMapped** returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, **CheckAccessFullyMapped** returns **FALSE**.

Return value

Type: **TemplateType**

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
--------	------	--------	----------	-------	---------

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

See also

[GatherCmpAlpha methods](#)

Texture2DArray::GatherCmpBlue methods

Article • 11/06/2019 • 2 minutes to read

Samples and compares a [Texture2DArray](#) and returns the blue component.

Overload list

Method	Description
GatherCmpBlue(S,float,float,int)	Samples and compares a texture and returns the blue component.
GatherCmpBlue(S,float,float,int,uint)	Samples and compares a texture and returns the blue component along with status about the operation.
GatherCmpBlue(S,float,float,int2,int2,int2,int2)	Samples and compares a texture and returns the blue component.
GatherCmpBlue(S,float,float,int2,int2,int2,int2,uint)	Samples and compares a texture and returns the blue component along with status about the operation.

See also

[Texture2DArray](#)

Texture2DArray::GatherCmpBlue(S,float, float,int) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their blue component against a compare value.

Syntax

Syntax

```
float4 GatherCmpBlue(  
    in SamplerComparisonState s,  
    in float3 location,  
    in float compare_value,  
    in int2 offset  
) ;
```

Parameters

s [in]

Type: **SamplerComparisonState**

The zero-based sampler index.

location [in]

Type: **float3**

The sample coordinates (u,v).

compare_value [in]

Type: **float**

A value to compare each against each sampled value.

offset [in]

Type: **int2**

An offset that is applied to the texture coordinate before sampling.

Return value

Type: **float4**

A four-component value, each component is the result of a per-component comparison.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmpBlue methods](#)

[Shader Model 5](#)

Texture2DArray::GatherCmpBlue(S,float, float,int2,int2,int2,int2) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their blue component against a compare value.

Syntax

Syntax

```
TemplateType GatherCmpBlue(
    in SamplerState S,
    in float      Location,
    in float      CompareValue,
    in int2       Offset1,
    in int2       Offset2,
    in int2       Offset3,
    in int2       Offset4
);
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

CompareValue [in]

Type: **float**

A value to compare each against each sampled value.

Offset1 [in]

Type: **int2**

The first offset component applied to the texture coordinates before sampling.

Offset2 [in]

Type: **int2**

The second offset component applied to the texture coordinates before sampling.

Offset3 [in]

Type: **int2**

The third offset component applied to the texture coordinates before sampling.

Offset4 [in]

Type: **int2**

The fourth offset component applied to the texture coordinates before sampling.

Return value

Type: **TemplateType**

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmpBlue methods](#)

Texture2DArray::GatherCmpBlue(S,float,float,int,uint) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their blue component against a compare value along with tile-mapping status.

Syntax

syntax

```
TemplateType GatherCmpBlue(  
    in SamplerState S,  
    in float      Location,  
    in float      CompareValue,  
    in int       Offset,  
    out uint     Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

CompareValue [in]

Type: **float**

A value to compare each against each sampled value.

Offset [in]

Type: **int**

The offset applied to the texture coordinates before sampling.

Status [out]

Type: `uint`

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns `TRUE` if all values from the corresponding `Sample`, `Gather`, or `Load` operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns `FALSE`.

Return value

Type: `TemplateType`

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmpBlue methods](#)

Texture2DArray::GatherCmpBlue(S,float,float,int2,int2,int2,int2,uint) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their blue component against a compare value along with tile-mapping status.

Syntax

syntax

```
TemplateType GatherCmpBlue(
    in SamplerState S,
    in float        Location,
    in float        CompareValue,
    in int2         Offset1,
    in int2         Offset2,
    in int2         Offset3,
    in int2         Offset4,
    out uint        Status
);
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

CompareValue [in]

Type: **float**

A value to compare each against each sampled value.

Offset1 [in]

Type: **int2**

The first offset component applied to the texture coordinates before sampling.

Offset2 [in]

Type: **int2**

The second offset component applied to the texture coordinates before sampling.

Offset3 [in]

Type: **int2**

The third offset component applied to the texture coordinates before sampling.

Offset4 [in]

Type: **int2**

The fourth offset component applied to the texture coordinates before sampling.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. **CheckAccessFullyMapped** returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, **CheckAccessFullyMapped** returns **FALSE**.

Return value

Type: **TemplateType**

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
--------	------	--------	----------	-------	---------

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

See also

[GatherCmpBlue methods](#)

Texture2DArray::GatherCmpGreen methods

Article • 11/06/2019 • 2 minutes to read

Samples and compares a [Texture2DArray](#) and returns the green component.

Overload list

Method	Description
GatherCmpGreen(S,float,float,int)	Samples and compares a texture and returns the green component.
GatherCmpGreen(S,float,float,int,uint)	Samples and compares a texture and returns the green component along with status about the operation.
GatherCmpGreen(S,float,float,int2,int2,int2,int2)	Samples and compares a texture and returns the green component.
GatherCmpGreen(S,float,float,int2,int2,int2,int2,uint)	Samples and compares a texture and returns the green component along with status about the operation.

See also

[Texture2DArray](#)

Texture2DArray::GatherCmpGreen(S,float,int) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their green component against a compare value.

Syntax

Syntax

```
float4 GatherCmpGreen(  
    in SamplerComparisonState s,  
    in float3 location,  
    in float compare_value,  
    in int2 offset  
) ;
```

Parameters

s [in]

Type: **SamplerComparisonState**

The zero-based sampler index.

location [in]

Type: **float3**

The sample coordinates (u,v).

compare_value [in]

Type: **float**

A value to compare each against each sampled value.

offset [in]

Type: **int2**

An offset that is applied to the texture coordinate before sampling.

Return value

Type: **float4**

A four-component value, each component is the result of a per-component comparison.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmpGreen methods](#)

[Shader Model 5](#)

Texture2DArray::GatherCmpGreen(S,float,int2,int2,int2,int2) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their green component against a compare value.

Syntax

Syntax

```
TemplateType GatherCmpGreen(  
    in SamplerState S,  
    in float      Location,  
    in float      CompareValue,  
    in int2       Offset1,  
    in int2       Offset2,  
    in int2       Offset3,  
    in int2       Offset4  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

CompareValue [in]

Type: **float**

A value to compare each against each sampled value.

Offset1 [in]

Type: **int2**

The first offset component applied to the texture coordinates before sampling.

Offset2 [in]

Type: **int2**

The second offset component applied to the texture coordinates before sampling.

Offset3 [in]

Type: **int2**

The third offset component applied to the texture coordinates before sampling.

Offset4 [in]

Type: **int2**

The fourth offset component applied to the texture coordinates before sampling.

Return value

Type: **TemplateType**

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmpGreen methods](#)

Texture2DArray::GatherCmpGreen(S,float,int,uint) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their green component against a compare value along with tile-mapping status.

Syntax

syntax

```
TemplateType GatherCmpGreen(  
    in SamplerState S,  
    in float      Location,  
    in float      CompareValue,  
    in int       Offset,  
    out uint     Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

CompareValue [in]

Type: **float**

A value to compare each against each sampled value.

Offset [in]

Type: **int**

The offset applied to the texture coordinates before sampling.

Status [out]

Type: `uint`

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns `TRUE` if all values from the corresponding `Sample`, `Gather`, or `Load` operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns `FALSE`.

Return value

Type: `TemplateType`

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmpGreen methods](#)

Texture2DArray::GatherCmpGreen(S,float,int2,int2,int2,int2,uint) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their green component against a compare value along with tile-mapping status.

Syntax

syntax

```
TemplateType GatherCmpGreen(
    in SamplerState S,
    in float          Location,
    in float          CompareValue,
    in int2           Offset1,
    in int2           Offset2,
    in int2           Offset3,
    in int2           Offset4,
    out uint          Status
);
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

CompareValue [in]

Type: **float**

A value to compare each against each sampled value.

Offset1 [in]

Type: **int2**

The first offset component applied to the texture coordinates before sampling.

Offset2 [in]

Type: **int2**

The second offset component applied to the texture coordinates before sampling.

Offset3 [in]

Type: **int2**

The third offset component applied to the texture coordinates before sampling.

Offset4 [in]

Type: **int2**

The fourth offset component applied to the texture coordinates before sampling.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. **CheckAccessFullyMapped** returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, **CheckAccessFullyMapped** returns **FALSE**.

Return value

Type: **TemplateType**

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
--------	------	--------	----------	-------	---------

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

See also

[GatherCmpGreen methods](#)

Texture2DArray::GatherCmpRed methods

Article • 11/06/2019 • 2 minutes to read

Samples and compares a [Texture2DArray](#) and returns the red component.

Overload list

Method	Description
GatherCmpRed(S,float,float,int)	Samples and compares a texture and returns the red component.
GatherCmpRed(S,float,float,int,uint)	Samples and compares a texture and returns the red component along with status about the operation.
GatherCmpRed(S,float,float,int2,int2,int2,int2)	Samples and compares a texture and returns the red component.
GatherCmpRed(S,float,float,int2,int2,int2,int2,uint)	Samples and compares a texture and returns the red component along with status about the operation.

See also

[Texture2DArray](#)

Texture2DArray::GatherCmpRed(S,float,f loat,int) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their red component against a compare value.

Syntax

Syntax

```
float4 GatherCmpRed(  
    in SamplerComparisonState s,  
    in float3 location,  
    in float compare_value,  
    in int2 offset  
) ;
```

Parameters

s [in]

Type: **SamplerComparisonState**

The zero-based sampler index.

location [in]

Type: **float3**

The sample coordinates (u,v).

compare_value [in]

Type: **float**

A value to compare each against each sampled value.

offset [in]

Type: **int2**

An offset that is applied to the texture coordinate before sampling.

Return value

Type: **float4**

A four-component value, each component is the result of a per-component comparison.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmpRed methods](#)

[Shader Model 5](#)

Texture2DArray::GatherCmpRed(S,float,float,int2,int2,int2,int2) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their red component against a compare value.

Syntax

Syntax

```
TemplateType GatherCmpRed(  
    in SamplerState S,  
    in float      Location,  
    in float      CompareValue,  
    in int2       Offset1,  
    in int2       Offset2,  
    in int2       Offset3,  
    in int2       Offset4  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

CompareValue [in]

Type: **float**

A value to compare each against each sampled value.

Offset1 [in]

Type: **int2**

The first offset component applied to the texture coordinates before sampling.

Offset2 [in]

Type: **int2**

The second offset component applied to the texture coordinates before sampling.

Offset3 [in]

Type: **int2**

The third offset component applied to the texture coordinates before sampling.

Offset4 [in]

Type: **int2**

The fourth offset component applied to the texture coordinates before sampling.

Return value

Type: **TemplateType**

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmpRed methods](#)

Texture2DArray::GatherCmpRed(S,float,float,int,uint) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their red component against a compare value along with tile-mapping status.

Syntax

syntax

```
TemplateType GatherCmpRed(
    in SamplerState S,
    in float        Location,
    in float        CompareValue,
    in int         Offset,
    out uint       Status
);
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

CompareValue [in]

Type: **float**

A value to compare each against each sampled value.

Offset [in]

Type: **int**

The offset applied to the texture coordinates before sampling.

Status [out]

Type: `uint`

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns `TRUE` if all values from the corresponding `Sample`, `Gather`, or `Load` operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns `FALSE`.

Return value

Type: `TemplateType`

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmpRed methods](#)

Texture2DArray::GatherCmpRed(S,float,float,int2,int2,int2,int2,uint) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their red component against a compare value along with tile-mapping status.

Syntax

syntax

```
TemplateType GatherCmpRed(  
    in SamplerState S,  
    in float        Location,  
    in float        CompareValue,  
    in int2         Offset1,  
    in int2         Offset2,  
    in int2         Offset3,  
    in int2         Offset4,  
    out uint        Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

CompareValue [in]

Type: **float**

A value to compare each against each sampled value.

Offset1 [in]

Type: **int2**

The first offset component applied to the texture coordinates before sampling.

Offset2 [in]

Type: **int2**

The second offset component applied to the texture coordinates before sampling.

Offset3 [in]

Type: **int2**

The third offset component applied to the texture coordinates before sampling.

Offset4 [in]

Type: **int2**

The fourth offset component applied to the texture coordinates before sampling.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. **CheckAccessFullyMapped** returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, **CheckAccessFullyMapped** returns **FALSE**.

Return value

Type: **TemplateType**

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
--------	------	--------	----------	-------	---------

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

See also

[GatherCmpRed methods](#)

Texture2DArray::GatherGreen methods

Article • 02/12/2021 • 2 minutes to read

Returns the green components of a [Texture2DArray](#)'s four texel values that would be used in a bi-linear filtering operation.

Overload list

Method	Description
GatherGreen(S,float,int)	Returns the green components of the four texel values that would be used in a bi-linear filtering operation.
GatherGreen(S,float,int,uint)	Returns the green components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.
GatherGreen(S,float,int2,int2,int2,int2)	Returns the green components of the four texel values that would be used in a bi-linear filtering operation.
GatherGreen(S,float,int2,int2,int2,int2,uint)	Returns the green components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

See also

[Texture2DArray](#)

Texture2DArray::GatherGreen(S,float,int) function

Article • 03/09/2021 • 2 minutes to read

Returns the green components of the four texel values that would be used in a bi-linear filtering operation.

Syntax

Syntax

```
TemplateType GatherGreen(  
    in sampler s,  
    in float3 location,  
    in int2 offset  
) ;
```

Parameters

s [in]

Type: **sampler**

The zero-based sampler index.

location [in]

Type: **float3**

The sample coordinates (u,v).

offset [in]

Type: **int2**

An offset that is applied to the texture coordinate before sampling.

Return value

Type: **TemplateType**

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherGreen methods](#)

[Shader Model 5](#)

Texture2DArray::GatherGreen(S,float,int2,int2,int2) function

Article • 03/06/2023 • 2 minutes to read

Returns the green components of the four texel values that would be used in a bi-linear filtering operation.

Syntax

Syntax

```
TemplateType GatherGreen(  
    in SamplerState S,  
    in float3      Location,  
    in int2        Offset1,  
    in int2        Offset2,  
    in int2        Offset3,  
    in int2        Offset4  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

Offset1 [in]

Type: **int2**

The first offset component applied to the texture coordinates before sampling.

Offset2 [in]

Type: **int2**

The second offset component applied to the texture coordinates before sampling.

Offset3 [in]

Type: `int2`

The third offset component applied to the texture coordinates before sampling.

Offset4 [in]

Type: `int2`

The fourth offset component applied to the texture coordinates before sampling.

Return value

Type: `TemplateType`

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherGreen methods](#)

Texture2DArray::GatherGreen(S,float,int,uint) function

Article • 03/06/2023 • 2 minutes to read

Returns the green components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

Syntax

Syntax

```
TemplateType GatherGreen(  
    in  SamplerState S,  
    in  float3      Location,  
    in  int         Offset,  
    out uint       Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

Offset [in]

Type: **int**

The offset applied to the texture coordinates before sampling.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns **FALSE**.

Return value

Type: [TemplateType](#)

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherGreen methods](#)

Texture2DArray::GatherGreen(S,float,int2,int2,int2,uint) function

Article • 03/06/2023 • 2 minutes to read

Returns the green components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

Syntax

Syntax

```
TemplateType GatherGreen(  
    in SamplerState S,  
    in float3      Location,  
    in int2        Offset1,  
    in int2        Offset2,  
    in int2        Offset3,  
    in int2        Offset4,  
    out uint       Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

Offset1 [in]

Type: **int2**

The first offset component applied to the texture coordinates before sampling.

Offset2 [in]

Type: **int2**

The second offset component applied to the texture coordinates before sampling.

Offset3 [in]

Type: **int2**

The third offset component applied to the texture coordinates before sampling.

Offset4 [in]

Type: **int2**

The fourth offset component applied to the texture coordinates before sampling.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. **CheckAccessFullyMapped** returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, **CheckAccessFullyMapped** returns **FALSE**.

Return value

Type: **TemplateType**

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherGreen methods](#)

Texture2DArray::GatherRed methods

Article • 02/12/2021 • 2 minutes to read

Returns the red components of a [Texture2DArray](#)'s four texel values that would be used in a bi-linear filtering operation.

Overload list

Method	Description
GatherRed(S,float,int)	Returns the red components of the four texel values that would be used in a bi-linear filtering operation.
GatherRed(S,float,int,uint)	Returns the red components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.
GatherRed(S,float,int2,int2,int2,int2)	Returns the red components of the four texel values that would be used in a bi-linear filtering operation.
GatherRed(S,float,int2,int2,int2,int2,uint)	Returns the red components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

See also

[Texture2DArray](#)

Texture2DArray::GatherRed(S,float,int) function

Article • 03/09/2021 • 2 minutes to read

Returns the red components of the four texel values that would be used in a bi-linear filtering operation.

Syntax

Syntax

```
TemplateType GatherRed(  
    in sampler s,  
    in float3 location,  
    in int2 offset  
) ;
```

Parameters

s [in]

Type: **sampler**

The zero-based sampler index.

location [in]

Type: **float3**

The sample coordinates (u,v).

offset [in]

Type: **int2**

An offset that is applied to the texture coordinate before sampling.

Return value

Type: **TemplateType**

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherRed methods](#)

[Shader Model 5](#)

Texture2DArray::GatherRed(S,float,int2,int2,int2,int2) function

Article • 03/06/2023 • 2 minutes to read

Returns the red components of the four texel values that would be used in a bi-linear filtering operation.

Syntax

Syntax

```
TemplateType GatherRed(  
    in SamplerState S,  
    in float3      Location,  
    in int2        Offset1,  
    in int2        Offset2,  
    in int2        Offset3,  
    in int2        Offset4  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

Offset1 [in]

Type: **int2**

The first offset component applied to the texture coordinates before sampling.

Offset2 [in]

Type: **int2**

The second offset component applied to the texture coordinates before sampling.

Offset3 [in]

Type: `int2`

The third offset component applied to the texture coordinates before sampling.

Offset4 [in]

Type: `int2`

The fourth offset component applied to the texture coordinates before sampling.

Return value

Type: `TemplateType`

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherRed methods](#)

Texture2DArray::GatherRed(S,float,int,uint) function

Article • 03/06/2023 • 2 minutes to read

Returns the red components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

Syntax

Syntax

```
TemplateType GatherRed(  
    in  SamplerState S,  
    in  float3      Location,  
    in  int         Offset,  
    out uint        Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

Offset [in]

Type: **int**

The offset applied to the texture coordinates before sampling.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns **FALSE**.

Return value

Type: [TemplateType](#)

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherRed methods](#)

Texture2DArray::GatherRed(S,float,int2,int2,int2,uint) function

Article • 03/06/2023 • 2 minutes to read

Returns the red components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

Syntax

Syntax

```
TemplateType GatherRed(  
    in SamplerState S,  
    in float3      Location,  
    in int2        Offset1,  
    in int2        Offset2,  
    in int2        Offset3,  
    in int2        Offset4,  
    out uint       Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

Offset1 [in]

Type: **int2**

The first offset component applied to the texture coordinates before sampling.

Offset2 [in]

Type: **int2**

The second offset component applied to the texture coordinates before sampling.

Offset3 [in]

Type: **int2**

The third offset component applied to the texture coordinates before sampling.

Offset4 [in]

Type: **int2**

The fourth offset component applied to the texture coordinates before sampling.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. **CheckAccessFullyMapped** returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, **CheckAccessFullyMapped** returns **FALSE**.

Return value

Type: **TemplateType**

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherRed methods](#)

Texture2DArray::GetDimensions function

Article • 03/09/2021 • 2 minutes to read

Returns the dimensions of the resource.

Syntax

syntax

```
void GetDimensions(  
    in  UINT MipLevel,  
    out UINT Width,  
    out UINT Height,  
    out UINT Elements,  
    out UINT NumberOfLevels  
) ;
```

Parameters

MipLevel [in]

Type: [UINT](#)

Optional. The mipmap level (must be specified if *NumberOfLevels* is used).

Width [out]

Type: [UINT](#)

The resource width, in texels.

Height [out]

Type: [UINT](#)

The resource height, in texels.

Elements [out]

Type: [UINT](#)

The number of elements in the array.

NumberOfLevels [out]

Type: **UINT**

The number of mipmap levels (requires *MipLevel* also).

Return value

Nothing

Remarks

This is a list of the overloaded versions of this method.

```
void GetDimensions(UINT MipLevel,
    out UINT Width,
    out UINT Height,
    out UINT Elements,
    out UINT NumberOfLevels);

void GetDimensions (out UINT Width,
    out float Height,
    out float Elements);

void GetDimensions(UINT MipLevel,
    out float Width,
    out float Height,
    out float Elements,
    out float NumberOfLevels);

void GetDimensions(out float Width,
    out float Height,
    out float Elements);
```

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	x

See also

Texture2DArray

Shader Model 5

Texture2DArray::Load methods

Article • 11/06/2019 • 2 minutes to read

Reads [Texture2DArray](#) data.

Overload list

Method	Description
Load(int,int)	Reads texture data.
Load(int,int,uint)	Reads texture data and returns status of the operation.

See also

[Texture2DArray](#)

Texture2DArray::Load(int,int,uint) function

Article • 03/09/2021 • 2 minutes to read

Reads texture data and returns status of the operation.

Syntax

syntax

```
Load(  
    in int Location,  
    in int Offset,  
    out uint Status  
);
```

Parameters

Location [in]

Type: **int**

The texture coordinates.

Offset [in]

Type: **int**

An offset applied to the texture coordinates before sampling.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns **TRUE** if all values from the corresponding [Sample](#), [Gather](#), or [Load](#) operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns **FALSE**.

Return value

Type:

The return type matches the type in the declaration for the [Texture2DArray](#) object.

See also

[Load methods](#)

Texture2DArray::mips.Operator function

Article • 03/09/2021 • 2 minutes to read

Returns a read-only resource variable.

Syntax

syntax

```
R mips.Operator[][](  
    in uint mipSlice,  
    in uint3 pos  
)
```

Parameters

mipSlice [in]

Type: **uint**

The mip slice index.

pos [in]

Type: **uint3**

The index position. The first and second components contain the (x, y) coordinates. The third component indicates the desired array slice.

Return value

Type: **R**

A read-only resource variable.

Remarks

Usage Example

```
Texture2DArray<float4> tex;  
uint mip = 2;  
uint2 pos_xy_and_array = {123, 456, 3};  
float4 f = tex.mips[mip][pos_xy_and_array];
```

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

See also

[Texture2DArray](#)

[Shader Model 5](#)

Texture2DArray::Operator function

Article • 03/09/2021 • 2 minutes to read

Returns a read-only resource variable.

Syntax

syntax

```
R Operator[](  
    in uint3 pos  
) ;
```

Parameters

pos [in]

Type: **uint3**

The index position. The first and second components contain the (x, y) coordinates. The third component indicates the desired array slice.

Return value

Type: **R**

A read-only resource variable.

Remarks

This method always accesses the first mip level. To specify other mip levels, use the [mip.operator\[\]\(\)](#) method instead.

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Texture2DArray](#)

[Shader Model 5](#)

Texture2DArray::Sample methods

Article • 11/06/2019 • 2 minutes to read

Samples a [Texture2DArray](#).

Overload list

Method	Description
Sample(S,float,int)	Samples a texture.
Sample(S,float,int,float)	Samples a texture with an optional value to clamp sample level-of-detail (LOD) values to.
Sample(S,float,int,float,uint)	Samples a texture with an optional value to clamp sample level-of-detail (LOD) values to, and returns status of the operation.

See also

[Texture2DArray](#)

Texture2DArray::Sample(S,float,int,float) function

Article • 03/09/2021 • 2 minutes to read

Samples a texture with an optional value to clamp sample level-of-detail (LOD) values to.

Syntax

Syntax

```
DXGI_FORMAT Sample(  
    in SamplerState S,  
    in float        Location,  
    in int         Offset,  
    in float        Clamp  
) ;
```

Parameters

S [in]

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

Offset [in]

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer miplevel; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Return value

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[Sample methods](#)

Texture2DArray::Sample(S,float,int,float,uint) function

Article • 03/09/2021 • 2 minutes to read

Samples a texture with an optional value to clamp sample level-of-detail (LOD) values to, and returns status of the operation.

Syntax

Syntax

```
DXGI_FORMAT Sample(  
    in SamplerState S,  
    in float        Location,  
    in int         Offset,  
    in float        Clamp,  
    out uint       Status  
) ;
```

Parameters

S [in]

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

Offset [in]

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer miplevel; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Status [out]

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns **TRUE** if all values from the corresponding [Sample](#), [Gather](#), or [Load](#) operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns **FALSE**.

Return value

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[Sample methods](#)

Texture2DArray::SampleBias methods

Article • 11/06/2019 • 2 minutes to read

Samples a [Texture2DArray](#), after applying the bias value to the mipmap level.

Overload list

Method	Description
SampleBias(S,float,float,int)	Samples a texture, after applying the bias value to the mipmap level.
SampleBias(S,float,float,int,float)	Samples a texture, after applying the bias value to the mipmap level, with an optional value to clamp sample level-of-detail (LOD) values to.
SampleBias(S,float,float,int,float,uint)	Samples a texture, after applying the bias value to the mipmap level, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

See also

[Texture2DArray](#)

SampleBias::SampleBias(S,float,float,int,float) function for Texture2DArray

Article • 03/15/2021 • 2 minutes to read

Samples a texture, after applying the bias value to the mipmap level, with an optional value to clamp sample level-of-detail (LOD) values to.

Syntax

Syntax

```
DXGI_FORMAT SampleBias(  
    in SamplerState S,  
    in float        Location,  
    in float        Bias,  
    in int         Offset,  
    in float        Clamp  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

Bias [in]

Type: **float**

The bias value, which is a floating-point number between 0.0 and 1.0 inclusive, is applied to a mip level before sampling.

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer miplevel; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

SampleBias methods

SampleBias::SampleBias(S,float,float,int,float,uint) function for Texture2DArray

Article • 03/15/2021 • 2 minutes to read

Samples a texture, after applying the bias value to the mipmap level, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

Syntax

syntax

```
DXGI_FORMAT SampleBias(  
    in SamplerState S,  
    in float        Location,  
    in float        Bias,  
    in int         Offset,  
    in float        Clamp,  
    out uint       Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3

Texture-Object Type	Parameter Type
TextureCubeArray	float4

Bias [in]

Type: **float**

The bias value, which is a floating-point number between 0.0 and 1.0 inclusive, is applied to a mip level before sampling.

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer miplevel; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns

TRUE if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, **CheckAccessFullyMapped** returns FALSE.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[SampleBias methods](#)

Texture2DArray::SampleCmp methods

Article • 11/06/2019 • 2 minutes to read

Samples a [Texture2DArray](#), using a comparison value to reject samples.

Overload list

Method	Description
SampleCmp(S,float,float,int)	Samples a texture, using a comparison value to reject samples.
SampleCmp(S,float,float,int,float)	Samples a texture, using a comparison value to reject samples, with an optional value to clamp sample level-of-detail (LOD) values to.
SampleCmp(S,float,float,int,float,uint)	Samples a texture, using a comparison value to reject samples, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

See also

[Texture2DArray](#)

SampleCmp::SampleCmp(S,float,float,int,float) function for Texture2DArray

Article • 03/15/2021 • 2 minutes to read

Samples a texture, using a comparison value to reject samples, with an optional value to clamp sample level-of-detail (LOD) values to.

Syntax

Syntax

```
DXGI_FORMAT SampleCmp(  
    in SamplerState S,  
    in float        Location,  
    in float        CompareValue,  
    in int          Offset,  
    in float        Clamp  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

CompareValue [in]

Type: **float**

A floating-point value to use as a comparison value.

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer miplevel; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[SampleCmp methods](#)

SampleCmp::SampleCmp(S,float,float,int,float,uint) function for Texture2DArray

Article • 03/15/2021 • 2 minutes to read

Samples a texture, using a comparison value to reject samples, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

Syntax

Syntax

```
DXGI_FORMAT SampleCmp(  
    in SamplerState S,  
    in float        Location,  
    in float        CompareValue,  
    in int          Offset,  
    in float        Clamp,  
    out uint        Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3

Texture-Object Type	Parameter Type
TextureCubeArray	float4

CompareValue [in]

Type: **float**

A floating-point value to use as a comparison value.

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer mipmap level; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns TRUE if all values from the corresponding [Sample](#), [Gather](#), or [Load](#) operation accessed

mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns `FALSE`.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[SampleCmp methods](#)

[Texture2DArray](#)

Texture2DArray::SampleCmpLevelZero methods

Article • 11/06/2019 • 2 minutes to read

Samples a [Texture2DArray](#) on mipmap level 0 only, using a comparison value to reject samples.

Overload list

Method	Description
SampleCmpLevelZero(S,float,float,int)	Samples a texture on mipmap level 0 only and compares the result to a comparison value.
SampleCmpLevelZero(S,float,float,int,uint)	Samples a texture on mipmap level 0 only and compares the result to a comparison value. Returns status about the operation.

See also

[Texture2DArray](#)

SampleCmpLevelZero::SampleCmpLevelZero(S,float,float,int,uint) function for Texture2DArray

Article • 03/15/2021 • 2 minutes to read

Samples a texture on mipmap level 0 only and compares the result to a comparison value. Returns status about the operation.

Syntax

```
syntax

DXGI_FORMAT SampleCmpLevelZero(
    in SamplerState S,
    in float          Location,
    in float          CompareValue,
    in int           Offset,
    out uint         Status
);
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3

Texture-Object Type	Parameter Type
TextureCubeArray	float4

CompareValue [in]

Type: **float**

A floating-point value to use as a comparison value.

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer mipmap level; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns **FALSE**.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[SampleCmpLevelZero methods](#)

[Texture2DArray](#)

Texture2DArray::SampleGrad methods

Article • 11/06/2019 • 2 minutes to read

Samples a [Texture2DArray](#) using a gradient to influence the way the sample location is calculated.

Overload list

Method	Description
SampleGrad(S,float,float,float,int)	Samples a texture, using a gradient to influence the way the sample location is calculated.
SampleGrad(S,float,float,float,int,float)	Samples a texture, using a gradient to influence the way the sample location is calculated, with an optional value to clamp sample level-of-detail (LOD) values to.
SampleGrad(S,float,float,float,int,float,uint)	Samples a texture, using a gradient to influence the way the sample location is calculated, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

See also

[Texture2DArray](#)

SampleGrad::SampleGrad(S,float,float, float,int,float) function for Texture2DArray

Article • 03/15/2021 • 2 minutes to read

Samples a texture, using a gradient to influence the way the sample location is calculated, with an optional value to clamp sample level-of-detail (LOD) values to.

Syntax

```
syntax

DXGI_FORMAT SampleGrad(
    in SamplerState S,
    in float          Location,
    in float          DDX,
    in float          DDY,
    in int           Offset,
    in float          Clamp
);
```

Parameters

S [in]

Type: [SamplerState](#)

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: [float](#)

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2

Texture-Object Type	Parameter Type
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

DDX [in]

Type: **float**

The rate of change of the surface geometry in the x direction. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	float
Texture2D, Texture2DArray	float2
Texture3D, TextureCube, TextureCubeArray	float3
Texture2DMS, Texture2DMSArray	not supported

DDY [in]

Type: **float**

The rate of change of the surface geometry in the y direction. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	float
Texture2D, Texture2DArray	float2
Texture3D, TextureCube, TextureCubeArray	float3
Texture2DMS, Texture2DMSArray	not supported

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer miplevel; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Return value

Type: **DXGI_FORMAT**

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[SampleGrad methods](#)

[Texture2DArray](#)

SampleGrad::SampleGrad(S,float,float,fl oat,int,float,uint) function for Texture2DArray

Article • 03/15/2021 • 2 minutes to read

Samples a texture, using a gradient to influence the way the sample location is calculated, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

Syntax

syntax

```
DXGI_FORMAT SampleGrad(  
    in SamplerState S,  
    in float        Location,  
    in float        DDX,  
    in float        DDY,  
    in int          Offset,  
    in float        Clamp,  
    out uint        Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float

Texture-Object Type	Parameter Type
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

DDX [in]

Type: **float**

The rate of change of the surface geometry in the x direction. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	float
Texture2D, Texture2DArray	float2
Texture3D, TextureCube, TextureCubeArray	float3
Texture2DMS, Texture2DMSArray	not supported

DDY [in]

Type: **float**

The rate of change of the surface geometry in the y direction. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	float
Texture2D, Texture2DArray	float2
Texture3D, TextureCube, TextureCubeArray	float3
Texture2DMS, Texture2DMSArray	not supported

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer miplevel; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns TRUE if all values from the corresponding [Sample](#), [Gather](#), or [Load](#) operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns FALSE.

Return value

Type: [**DXGI_FORMAT**](#)

The texture format, which is one of the typed values listed in [**DXGI_FORMAT**](#).

See also

SampleGrad methods

[Texture2DArray](#)

Texture2DArray::SampleLevel methods

Article • 11/06/2019 • 2 minutes to read

Samples a [Texture2DArray](#) on the specified mipmap level.

Overload list

Method	Description
SampleLevel(S,float,float,int)	Samples a texture on the specified mipmap level.
SampleLevel(S,float,float,int,uint)	Samples a texture on the specified mipmap level and returns status about the operation.

See also

[Texture2DArray](#)

SampleLevel::SampleLevel(S,float,float,int,uint) function for Texture2DArray

Article • 03/15/2021 • 2 minutes to read

Samples a texture on the specified mipmap level and returns status about the operation.

Syntax

syntax

```
DXGI_FORMAT SampleLevel(  
    in SamplerState S,  
    in float        Location,  
    in float        LOD,  
    in int         Offset,  
    out uint       Status  
)
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

LOD [in]

Type: **float**

[in] A number that specifies the mipmap level. If the value is ≤ 0 , mipmap level 0 (biggest map) is used. The fractional value (if supplied) is used to interpolate between two mipmap levels.

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer miplevel; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns **FALSE**.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[SampleLevel methods](#)

[Texture2DArray](#)

Texture3D

Article • 10/24/2019 • 2 minutes to read

Texture3D type ([as it exists in Shader Model 4](#)) plus resource variables. This texture object supports the following methods in addition to the methods in Shader Model 4.

Method	Description
GetDimensions	Gets the resource dimensions.
Load	Reads texture data.
mips.Operator[] []	Gets a read-only resource variable.
Operator[]	Gets a read-only resource variable.
Sample	Samples a texture.
SampleBias	Samples a texture, after applying the bias value to the mipmap level.
SampleGrad	Samples a texture using a gradient to influence the way the sample location is calculated.
SampleLevel	Samples a texture on the specified mipmap level.

Minimum Shader Model

This object is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This object is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Shader Model 5 Objects](#)

Texture3D::GetDimensions function

Article • 03/09/2021 • 2 minutes to read

Returns the dimensions of the resource.

Syntax

syntax

```
void GetDimensions(  
    in  UINT MipLevel,  
    out UINT Width,  
    out UINT Height,  
    out UINT Depth,  
    out UINT NumberOfLevels  
);
```

Parameters

MipLevel [in]

Type: [UINT](#)

Optional. Mipmap level (must be specified if *NumberOfLevels* is used).

Width [out]

Type: [UINT](#)

The resource width, in texels.

Height [out]

Type: [UINT](#)

The resource height, in texels.

Depth [out]

Type: [UINT](#)

The resource depth, in texels.

NumberOfLevels [out]

Type: [UINT](#)

The number of mipmap levels (requires *MipLevel* also).

Return value

Nothing

Remarks

This is a list of the overloaded versions of this method.

```
void GetDimensions(UINT MipLevel,
    out UINT Width,
    out UINT Height,
    out UINT Depth,
    out UINT NumberOfLevels);

void GetDimensions (out UINT Width,
    out UINT Height,
    out UINT Depth);

void GetDimensions(UINT MipLevel,
    out float Width,
    out float Height,
    out float Depth,
    out float NumberOfLevels);

void GetDimensions(out float Width,
    out float Height,
    out float Depth);
```

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

See also

[Texture3D](#)

Shader Model 5

Texture3D::Load methods

Article • 03/09/2021 • 2 minutes to read

Reads texture data.

Overload list

Method	Description
Load(int,int)	Reads texture data.
Load(int,int,uint)	Reads texture data and returns status of the operation.

See also

[Texture3D](#)

Texture3D::Load(int,int,uint) function

Article • 03/09/2021 • 2 minutes to read

Reads texture data and returns status of the operation.

Syntax

syntax

```
Load(  
    in int Location,  
    in int Offset,  
    out uint Status  
) ;
```

Parameters

Location [in]

Type: **int**

The texture coordinates.

Offset [in]

Type: **int**

An offset applied to the texture coordinates before sampling.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns **TRUE** if all values from the corresponding `Sample`, `Gather`, or `Load` operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns **FALSE**.

Return value

Type:

The return type matches the type in the declaration for the [Texture3D](#) object.

See also

[Load methods](#)

[Texture3D](#)

Texture3D::mips.Operator function

Article • 03/09/2021 • 2 minutes to read

Returns a read-only resource variable.

Syntax

syntax

```
R mips.Operator[][](  
    in uint mipSlice,  
    in uint3 pos  
)
```

Parameters

mipSlice [in]

Type: **uint**

The mip slice index.

pos [in]

Type: **uint3**

The index position. Contains the (x, y, z) coordinates.

Return value

Type: **R**

A read-only resource variable.

Remarks

Usage Example

```
Texture3D<float4> tex;  
uint mip = 2;  
uint3 pos_xyz = {123, 456, 789};  
float4 f = tex.mips[mip][pos_xyz];
```

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

See also

[Texture3D](#)

[Shader Model 5](#)

Texture3D::Operator function

Article • 03/09/2021 • 2 minutes to read

Returns a read-only resource variable.

Syntax

syntax

```
R Operator[](  
    in uint3 pos  
) ;
```

Parameters

pos [in]

Type: **uint3**

The index position. Contains the (x, y, z) coordinates.

Return value

Type: **R**

A read-only resource variable.

Remarks

This method always accesses the first mip level. To specify other mip levels use the [mip.operator\[\]\[\]](#) instead.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Texture3D](#)

[Shader Model 5](#)

Texture3D::Sample methods

Article • 03/09/2021 • 2 minutes to read

Samples a texture.

Overload list

Method	Description
Sample(S,float,int)	Samples a texture.
Sample(S,float,int,float)	Samples a texture with an optional value to clamp sample level-of-detail (LOD) values to.
Sample(S,float,int,float,uint)	Samples a texture with an optional value to clamp sample level-of-detail (LOD) values to, and returns status of the operation.

See also

[Texture3D](#)

[Texture-Object](#)

Texture3D::Sample(S,float,int,float) function

Article • 03/09/2021 • 2 minutes to read

Samples a texture with an optional value to clamp sample level-of-detail (LOD) values to.

Syntax

Syntax

```
DXGI_FORMAT Sample(  
    in SamplerState S,  
    in float        Location,  
    in int          Offset,  
    in float        Clamp  
) ;
```

Parameters

S [in]

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

Offset [in]

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer miplevel; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Return value

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[Sample methods](#)

[Texture3D](#)

Texture3D::Sample(S,float,int,float,uint) function

Article • 03/09/2021 • 2 minutes to read

Samples a texture with an optional value to clamp sample level-of-detail (LOD) values to, and returns status of the operation.

Syntax

Syntax

```
DXGI_FORMAT Sample(  
    in SamplerState S,  
    in float        Location,  
    in int         Offset,  
    in float        Clamp,  
    out uint       Status  
) ;
```

Parameters

S [in]

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

Offset [in]

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer miplevel; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Status [out]

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns **FALSE**.

Return value

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[Sample methods](#)

[Texture3D](#)

Texture3D::SampleBias methods

Article • 03/09/2021 • 2 minutes to read

Samples a texture, after applying the bias value to the mipmap level.

Overload list

Method	Description
SampleBias(S,float,float,int)	Samples a texture, after applying the bias value to the mipmap level.
SampleBias(S,float,float,int,float)	Samples a texture, after applying the bias value to the mipmap level, with an optional value to clamp sample level-of-detail (LOD) values to.
SampleBias(S,float,float,int,float,uint)	Samples a texture, after applying the bias value to the mipmap level, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

See also

[Texture3D](#)

[Texture-Object](#)

SampleBias::SampleBias(S,float,float,int,float) function for Texture3D

Article • 04/02/2021 • 2 minutes to read

Samples a texture, after applying the bias value to the mipmap level, with an optional value to clamp sample level-of-detail (LOD) values to.

Syntax

Syntax

```
DXGI_FORMAT SampleBias(  
    in SamplerState S,  
    in float        Location,  
    in float        Bias,  
    in int         Offset,  
    in float        Clamp  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

Bias [in]

Type: **float**

The bias value, which is a floating-point number between 0.0 and 1.0 inclusive, is applied to a mip level before sampling.

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer miplevel; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

SampleBias methods

Texture3D

SampleBias::SampleBias(S,float,float,int,float,uint) function for Texture3D

Article • 03/15/2021 • 2 minutes to read

Samples a texture, after applying the bias value to the mipmap level, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

Syntax

syntax

```
DXGI_FORMAT SampleBias(  
    in SamplerState S,  
    in float        Location,  
    in float        Bias,  
    in int         Offset,  
    in float        Clamp,  
    out uint       Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3

Texture-Object Type	Parameter Type
TextureCubeArray	float4

Bias [in]

Type: **float**

The bias value, which is a floating-point number between 0.0 and 1.0 inclusive, is applied to a mip level before sampling.

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer miplevel; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns

TRUE if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, **CheckAccessFullyMapped** returns FALSE.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[SampleBias methods](#)

[Texture3D](#)

Texture3D::SampleGrad methods

Article • 03/09/2021 • 2 minutes to read

Samples a texture using a gradient to influence the way the sample location is calculated.

Overload list

Method	Description
SampleGrad(S,float,float,float,int)	Samples a texture, using a gradient to influence the way the sample location is calculated.
SampleGrad(S,float,float,float,int,float)	Samples a texture, using a gradient to influence the way the sample location is calculated, with an optional value to clamp sample level-of-detail (LOD) values to.
SampleGrad(S,float,float,float,int,float,uint)	Samples a texture, using a gradient to influence the way the sample location is calculated, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

See also

[Texture3D](#)

[Texture-Object](#)

SampleGrad::SampleGrad(S,float,float, float,int,float) function for Texture3D

Article • 03/15/2021 • 2 minutes to read

Samples a texture, using a gradient to influence the way the sample location is calculated, with an optional value to clamp sample level-of-detail (LOD) values to.

Syntax

Syntax

```
DXGI_FORMAT SampleGrad(  
    in SamplerState S,  
    in float        Location,  
    in float        DDX,  
    in float        DDY,  
    in int          Offset,  
    in float        Clamp  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3

Texture-Object Type	Parameter Type
TextureCubeArray	float4

DDX [in]

Type: **float**

The rate of change of the surface geometry in the x direction. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	float
Texture2D, Texture2DArray	float2
Texture3D, TextureCube, TextureCubeArray	float3
Texture2DMS, Texture2DMSArray	not supported

DDY [in]

Type: **float**

The rate of change of the surface geometry in the y direction. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	float
Texture2D, Texture2DArray	float2
Texture3D, TextureCube, TextureCubeArray	float3
Texture2DMS, Texture2DMSArray	not supported

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer miplevel; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Return value

Type: **DXGI_FORMAT**

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[SampleGrad methods](#)

[Texture3D](#)

SampleGrad::SampleGrad(S,float,float,fl oat,int,float,uint) function for Texture3D

Article • 03/15/2021 • 2 minutes to read

Samples a texture, using a gradient to influence the way the sample location is calculated, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

Syntax

syntax

```
DXGI_FORMAT SampleGrad(  
    in SamplerState S,  
    in float        Location,  
    in float        DDX,  
    in float        DDY,  
    in int          Offset,  
    in float        Clamp,  
    out uint        Status  
) ;
```

Parameters

S [in]

Type: [SamplerState](#)

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: [float](#)

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2

Texture-Object Type	Parameter Type
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

DDX [in]

Type: **float**

The rate of change of the surface geometry in the x direction. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	float
Texture2D, Texture2DArray	float2
Texture3D, TextureCube, TextureCubeArray	float3
Texture2DMS, Texture2DMSArray	not supported

DDY [in]

Type: **float**

The rate of change of the surface geometry in the y direction. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	float
Texture2D, Texture2DArray	float2
Texture3D, TextureCube, TextureCubeArray	float3
Texture2DMS, Texture2DMSArray	not supported

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer miplevel; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns **TRUE** if all values from the corresponding [Sample](#), [Gather](#), or [Load](#) operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns **FALSE**.

Return value

Type: [**DXGI_FORMAT**](#)

The texture format, which is one of the typed values listed in [**DXGI_FORMAT**](#).

See also

[SampleGrad methods](#)

Texture3D

Texture3D::SampleLevel methods

Article • 03/09/2021 • 2 minutes to read

Samples a texture on the specified mipmap level.

Overload list

Method	Description
SampleLevel(S,float,float,int)	Samples a texture on the specified mipmap level.
SampleLevel(S,float,float,int,uint)	Samples a texture on the specified mipmap level and returns status about the operation.

See also

[Texture3D](#)

[Texture-Object](#)

SampleLevel::SampleLevel(S,float,float,int,uint) function for Texture3D

Article • 03/15/2021 • 2 minutes to read

Samples a texture on the specified mipmap level and returns status about the operation.

Syntax

syntax

```
DXGI_FORMAT SampleLevel(  
    in SamplerState S,  
    in float        Location,  
    in float        LOD,  
    in int         Offset,  
    out uint       Status  
)
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

LOD [in]

Type: **float**

[in] A number that specifies the mipmap level. If the value is ≤ 0 , mipmap level 0 (biggest map) is used. The fractional value (if supplied) is used to interpolate between two mipmap levels.

Offset [in]

Type: **int**

An optional texture coordinate offset, which can be used for any texture-object type; the offset is applied to the location before sampling. Use an offset only at an integer miplevel; otherwise, you may get results that do not translate well to hardware. The argument type is dependent on the texture-object type. For more info, see [Applying Integer Offsets](#).

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	int
Texture2D, Texture2DArray	int2
Texture3D	int3
TextureCube, TextureCubeArray	not supported

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns **FALSE**.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[SampleLevel methods](#)

[Texture3D](#)

Texture2DMS

Article • 10/24/2019 • 2 minutes to read

Texture2DMS type (as it exists in Shader Model 4) plus resource variables.

In this section

Topic	Description
GetDimensions	Returns the dimensions of the resource.
GetSamplePosition	Returns the sample position for the sample index provided.
Load methods	Retrieves a value from the resource at the location and sample index provided.
sample.Operator[] []	Retrieves a value from the resource at the location and sample index provided.
Operator[]	Retrieves a value from the resource at the location provided at sample index 0.

Minimum Shader Model

This object is supported in the following shader models.

Shader Model	Supported
Shader model 4 and higher	yes

This object is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Shader Model 5 Objects](#)

Texture2DMS::GetDimensions function

Article • 03/09/2021 • 2 minutes to read

Returns the dimensions of the resource.

Syntax

syntax

```
void GetDimensions(  
    out UINT Width,  
    out UINT Height,  
    out UINT NumberOfSamples  
) ;
```

Parameters

Width [out]

Type: [UINT](#)

The resource width, in texels.

Height [out]

Type: [UINT](#)

The resource height, in texels.

NumberOfSamples [out]

Type: [UINT](#)

The number of sample locations.

Return value

This function does not return a value.

Remarks

This is a list of the overloaded versions of this method.

```
void GetDimensions(out float Width,  
    out float Height,  
    out float NumberOfSamples);
```

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

See also

[Texture2DMS](#)

[Shader Model 5](#)

Texture2DMS::GetSamplePosition function

Article • 12/10/2020 • 2 minutes to read

Returns the sample position for the sample index provided.

Syntax

syntax

```
float2 GetSamplePosition(  
    in int sampleindex  
) ;
```

Parameters

sampleindex [in]

Type: [int](#)

The zero-based index of a sample location.

Return value

Type: [float2](#)

A sample location.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

Texture2DMS

Shader Model 5

Texture2DMS::Load methods

Article • 03/09/2021 • 2 minutes to read

Retrieves a value from the resource at the location and sample index provided.

Overload list

Method	Description
Load(int,int)	Reads texture data.
Load(int,int,int)	Reads texture data.
Load(int,int,int,uint)	Reads texture data and returns status of the operation.

See also

[Texture2DMS](#)

Texture2DMS::Load(int,int) function

Article • 03/09/2021 • 2 minutes to read

Gets one value.

Syntax

syntax

```
T Load(  
    in int2 coord,  
    in int sampleindex  
);
```

Parameters

coord [in]

Type: **int2**

The input location.

sampleindex [in]

Type: **int**

The sample index.

Return value

Type: **T**

One value, whose type matches the texture template type.

Remarks

This is a list of the overloaded versions of this method.

```
void Load(in int2 coord,  
          in int sampleindex,
```

```
    in int2 offset);
```

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	x

See also

[Load methods](#)

[Shader Model 5](#)

Texture2DMS::Load(int,int,int,uint) function

Article • 03/09/2021 • 2 minutes to read

Reads texture data and returns status of the operation.

Syntax

syntax

```
Load(  
    in int2 Location,  
    in int sampleindex,  
    in int2 Offset,  
    out uint Status  
) ;
```

Parameters

Location [in]

Type: **int2**

The texture coordinates.

sampleindex [in]

Type: **int**

The sample index.

Offset [in]

Type: **int2**

An offset applied to the texture coordinates before loading.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns

TRUE if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, **CheckAccessFullyMapped** returns **FALSE**.

Return value

Type:

The return type matches the type in the declaration for the [Texture2DMS](#) object.

See also

[Load methods](#)

[Texture2DMS](#)

Texture2DMS::sample.Operator function

Article • 03/09/2021 • 2 minutes to read

Retrieves a value from the resource at the location and sample index provided.

Syntax

syntax

```
R sample.Operator[][](  
    in uint sampleSlice,  
    in uint2 pos  
)
```

Parameters

sampleSlice [in]

Type: **uint**

The sample slice index.

pos [in]

Type: **uint2**

The index position. The components contain the (x, y) coordinates.

Return value

Type: **R**

A read-only resource variable.

Remarks

Usage Example

```
Texture2DMS<float4, 8> tex;

float4 main( float3 tcoord : texturecoord0 ) : SV_Target
{
    return s_msTexture.sample[2][tcoord];
}
```

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Texture2DMS](#)

[Shader Model 5](#)

Texture2DMS::Operator function

Article • 12/10/2020 • 2 minutes to read

Retrieves a value from the resource at the location provided at sample index 0.

Syntax

syntax

```
R Operator[](  
    in uint2 pos  
) ;
```

Parameters

pos [in]

Type: **uint2**

The index position. Contains the (x, y) coordinates.

Return value

Type: **R**

A read-only resource variable.

Remarks

To select a particular sample, refer to [sample.Operator\[\]\[\]](#).

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

Texture2DMS

Shader Model 5

Texture2DMSArray

Article • 02/12/2021 • 2 minutes to read

Texture2DMSArray type (as it exists in Shader Model 4) plus resource variables.

Method	Description
GetDimensions	Gets the resource dimensions.
GetSamplePosition	Gets the position of the specified sample.
Load	Gets one value.
sample.Operator[]()	Gets a read-only resource variable.

Minimum Shader Model

This object is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This object is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

See also

[Shader Model 5 Objects](#)

Texture2DMSArray::GetDimensions function

Article • 03/09/2021 • 2 minutes to read

Returns the dimensions of the resource.

Syntax

syntax

```
void GetDimensions(  
    out UINT Width,  
    out UINT Height,  
    out UINT Elements,  
    out UINT NumberOfSamples  
) ;
```

Parameters

Width [out]

Type: [UINT](#)

The resource width, in texels.

Height [out]

Type: [UINT](#)

The resource height, in texels.

Elements [out]

Type: [UINT](#)

The number of elements in the array.

NumberOfSamples [out]

Type: [UINT](#)

The number of sample locations.

Return value

Nothing

Remarks

This is a list of the overloaded versions of this method.

```
void GetDimensions(out float Width,  
                   out float Height,  
                   out float Elements,  
                   out float NumberOfSamples);
```

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

See also

[Texture2DMSArray](#)

[Shader Model 5](#)

Texture2DMSArray::GetSamplePosition function

Article • 03/09/2021 • 2 minutes to read

Gets the position of the specified sample.

Syntax

syntax

```
float2 GetSamplePosition(  
    in int sampleindex  
) ;
```

Parameters

sampleindex [in]

Type: [int](#)

The zero-based index of a sample location.

Return value

Type: [float2](#)

A sample location.

Remarks

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

Texture2DMSArray

Shader Model 5

Texture2DMSArray::Load methods

Article • 03/09/2021 • 2 minutes to read

Reads texture data.

Overload list

Method	Description
Load(int,int)	Reads texture data.
Load(int,int,int)	Reads texture data.
Load(int,int,int,uint)	Reads texture data and returns status of the operation.

See also

[Texture2DMSArray](#)

Texture2DMSArray::Load(int,int) function

Article • 03/09/2021 • 2 minutes to read

Gets one value.

Syntax

syntax

```
T Load(  
    in int3 coord,  
    in int sampleindex  
) ;
```

Parameters

coord [in]

Type: **int3**

The input location.

sampleindex [in]

Type: **int**

The sample index.

Return value

Type: **T**

One value, whose type matches the texture template type.

Remarks

This is a list of the overloaded versions of this method.

```
void Load(in int3 coord,  
         in int sampleindex,  
         in int2 offset);
```

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

See also

[Load methods](#)

[Shader Model 5](#)

Texture2DMSArray::Load(int,int,int,uint) function

Article • 03/09/2021 • 2 minutes to read

Reads texture data and returns status of the operation.

Syntax

syntax

```
Load(  
    in int Location,  
    in int sampleindex,  
    in int Offset,  
    out uint Status  
) ;
```

Parameters

Location [in]

Type: **int**

The texture coordinates.

sampleindex [in]

Type: **int**

The sample index.

Offset [in]

Type: **int**

An offset applied to the texture coordinates before sampling.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns

TRUE if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, **CheckAccessFullyMapped** returns **FALSE**.

Return value

Type:

The return type matches the type in the declaration for the [Texture2DMSArray](#) object.

See also

[Load methods](#)

[Texture2DMSArray](#)

Texture2DMSArray::sample.Operator function

Article • 03/09/2021 • 2 minutes to read

Returns a read-only resource variable.

Syntax

syntax

```
R sample.Operator[][](  
    in uint sampleSlice,  
    in uint3 pos  
)
```

Parameters

sampleSlice [in]

Type: **uint**

The sample slice index.

pos [in]

Type: **uint3**

The index position. The first and second components contain the (x, y) coordinates. The third component indicates the desired array slice.

Return value

Type: **R**

A read-only resource variable.

Remarks

Usage Example

```
Texture2DMSArray<float4, 4> alpha;  
  
float4 main( float3 tcoord : texturecoord0 ) : SV_Target  
{  
    return s_msTexture.sample[2][tcoord];  
}
```

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

See also

[Texture2DMSArray](#)

[Shader Model 5](#)

TextureCube object

Article • 02/12/2021 • 2 minutes to read

TextureCube type (as it exists in Shader Model 4) plus resource variables. This texture object supports these methods in addition to the methods in Shader Model 4.

- [Methods](#)

Methods

The **TextureCube** object has these methods.

Method	Description
Gather	Returns the four texel values that would be used in a bi-linear filtering operation.
GatherAlpha	Returns the alpha components of the four texel values that would be used in a bi-linear filtering operation.
GatherBlue	Returns the blue components of the four texel values that would be used in a bi-linear filtering operation.
GatherCmp	For four texel values that would be used in a bi-linear filtering operation, returns their comparison against a compare value.
GatherCmpAlpha	For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their alpha component against a compare value.
GatherCmpBlue	For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their blue component against a compare value.
GatherCmpGreen	For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their green component against a compare value.
GatherCmpRed	For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their red component against a compare value.
GatherGreen	Returns the green components of the four texel values that would be used in a bi-linear filtering operation.
GatherRed	Returns the red components of the four texel values that would be used in a bi-linear filtering operation.
Sample	Samples a texture.
SampleBias	Samples a texture, after applying the bias value to the mipmap level.

Method	Description
SampleCmp	Samples a texture, using a comparison value to reject samples.
SampleCmpLevelZero	Samples a texture (mipmap level 0 only), using a comparison value to reject samples.
SampleGrad	Samples a texture using a gradient to influence the way the sample location is calculated.
SampleLevel	Samples a texture on the specified mipmap level.

Remarks

Minimum Shader Model

This object is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This object is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Shader Model 5 Objects](#)

TextureCube::Gather methods

Article • 03/09/2021 • 2 minutes to read

Returns the four texel values that would be used in a bi-linear filtering operation.

See the documentation on [gather4](#) for more information describing the underlying DXBC instruction.

Overload list

Method	Description
Gather(S,float)	Samples a texture and returns the four samples (red component only) that are used for bilinear interpolation.
Gather(S,float,uint)	Samples a texture and returns all four components along with status about the operation.

See also

[TextureCube](#)

TextureCube::Gather(S,float,uint) function

Article • 03/06/2023 • 2 minutes to read

Returns the four texel values that would be used in a bi-linear filtering operation.

Syntax

syntax

```
TemplateType Gather(  
    in SamplerState S,  
    in float3      Location,  
    out uint       Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. **CheckAccessFullyMapped** returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, **CheckAccessFullyMapped** returns **FALSE**.

Return value

Type: **TemplateType**

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Gather methods](#)

[TextureCube](#)

TextureCube::GatherAlpha methods

Article • 03/09/2021 • 2 minutes to read

Returns the alpha components of the four texel values that would be used in a bi-linear filtering operation.

Overload list

Method	Description
GatherAlpha(S,float,uint)	Returns the alpha components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

See also

[TextureCube](#)

TextureCube::GatherAlpha(S,float,uint) function

Article • 03/06/2023 • 2 minutes to read

Returns the alpha components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

Syntax

Syntax

```
TemplateType GatherAlpha(  
    in SamplerState S,  
    in float3      Location,  
    out uint       Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns **TRUE** if all values from the corresponding [Sample](#), [Gather](#), or [Load](#) operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns **FALSE**.

Return value

Type: [TemplateType](#)

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherAlpha methods](#)

[TextureCube](#)

TextureCube::GatherBlue methods

Article • 03/09/2021 • 2 minutes to read

Returns the blue components of the four texel values that would be used in a bi-linear filtering operation.

Overload list

Method	Description
GatherBlue(S,float,uint)	Returns the blue components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

See also

[TextureCube](#)

TextureCube::GatherBlue(S,float,uint) function

Article • 03/06/2023 • 2 minutes to read

Returns the blue components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

Syntax

Syntax

```
TemplateType GatherBlue(  
    in SamplerState S,  
    in float3      Location,  
    out uint       Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns **TRUE** if all values from the corresponding [Sample](#), [Gather](#), or [Load](#) operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns **FALSE**.

Return value

Type: [TemplateType](#)

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherBlue methods](#)

[TextureCube](#)

TextureCube::GatherCmp methods

Article • 03/09/2021 • 2 minutes to read

For four texel values of a [TextureCube](#) that would be used in a bi-linear filtering operation, returns their comparison against a compare value.

See the documentation on [gather4_c](#) for more information describing the underlying DXBC instruction.

Overload list

Method	Description
GatherCmp(S,float,float,uint)	Samples and compares a texture and returns all four components along with status about the operation.

See also

[TextureCube](#)

TextureCube::GatherCmp(S,float,float,uint) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns their comparison against a compare value.

Syntax

Syntax

```
TemplateType GatherCmp(  
    in SamplerState S,  
    in float        Location,  
    in float        CompareValue,  
    out uint        Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

CompareValue [in]

Type: **float**

A value to compare each against each sampled value.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns **FALSE**.

Return value

Type: [TemplateType](#)

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmp methods](#)

[TextureCube](#)

TextureCube::GatherCmpAlpha methods

Article • 03/09/2021 • 2 minutes to read

Samples and compares a texture and returns the alpha component.

Overload list

Method	Description
GatherCmpAlpha(S,float,float,uint)	Samples and compares a texture and returns the alpha component along with status about the operation.

See also

[TextureCube](#)

TextureCube::GatherCmpAlpha(S,float,uint) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their alpha component against a compare value along with tile-mapping status.

Syntax

syntax

```
TemplateType GatherCmpAlpha(  
    in SamplerState S,  
    in float      Location,  
    in float      CompareValue,  
    out uint      Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

CompareValue [in]

Type: **float**

A value to compare each against each sampled value.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns **FALSE**.

Return value

Type: [TemplateType](#)

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmpAlpha methods](#)

[TextureCube](#)

TextureCube::GatherCmpBlue methods

Article • 03/09/2021 • 2 minutes to read

Samples and compares a texture and returns the blue component.

Overload list

Method	Description
GatherCmpBlue(S,float,float,uint)	Samples and compares a texture and returns the blue component along with status about the operation.

See also

[TextureCube](#)

TextureCube::GatherCmpBlue(S,float,flo at,uint) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their blue component against a compare value along with tile-mapping status.

Syntax

syntax

```
TemplateType GatherCmpBlue(  
    in SamplerState S,  
    in float      Location,  
    in float      CompareValue,  
    out uint      Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

CompareValue [in]

Type: **float**

A value to compare each against each sampled value.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns **FALSE**.

Return value

Type: [TemplateType](#)

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmpBlue methods](#)

[TextureCube](#)

TextureCube::GatherCmpGreen methods

Article • 03/09/2021 • 2 minutes to read

Samples and compares a texture and returns the green component.

Overload list

Method	Description
GatherCmpGreen(S,float,float,uint)	Samples and compares a texture and returns the green component along with status about the operation.

See also

[TextureCube](#)

TextureCube::GatherCmpGreen(S,float,uint) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their green component against a compare value along with tile-mapping status.

Syntax

syntax

```
TemplateType GatherCmpGreen(  
    in SamplerState S,  
    in float      Location,  
    in float      CompareValue,  
    out uint      Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

CompareValue [in]

Type: **float**

A value to compare each against each sampled value.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns **FALSE**.

Return value

Type: [TemplateType](#)

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmpGreen methods](#)

[TextureCube](#)

TextureCube::GatherCmpRed methods

Article • 03/09/2021 • 2 minutes to read

Samples and compares a texture and returns the red component.

Overload list

Method	Description
GatherCmpRed(S,float,float,uint)	Samples and compares a texture and returns the red component along with status about the operation.

See also

[TextureCube](#)

TextureCube::GatherCmpRed(S,float,floa t,uint) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their red component against a compare value along with tile-mapping status.

Syntax

syntax

```
TemplateType GatherCmpRed(  
    in SamplerState S,  
    in float      Location,  
    in float      CompareValue,  
    out uint      Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

CompareValue [in]

Type: **float**

A value to compare each against each sampled value.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns **FALSE**.

Return value

Type: [TemplateType](#)

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmpRed methods](#)

[TextureCube](#)

TextureCube::GatherGreen methods

Article • 03/09/2021 • 2 minutes to read

Returns the green components of the four texel values that would be used in a bi-linear filtering operation.

Overload list

Method	Description
GatherGreen(S,float,uint)	Returns the green components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

See also

[TextureCube](#)

TextureCube::GatherGreen(S,float,uint) function

Article • 03/06/2023 • 2 minutes to read

Returns the green components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

Syntax

Syntax

```
TemplateType GatherGreen(  
    in SamplerState S,  
    in float3      Location,  
    out uint       Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns **TRUE** if all values from the corresponding [Sample](#), [Gather](#), or [Load](#) operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns **FALSE**.

Return value

Type: **TemplateType**

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherGreen methods](#)

[TextureCube](#)

TextureCube::GatherRed methods

Article • 03/09/2021 • 2 minutes to read

Returns the red components of the four texel values that would be used in a bi-linear filtering operation.

Overload list

Method	Description
GatherRed(S,float,uint)	Returns the red components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

See also

[TextureCube](#)

TextureCube::GatherRed(S,float,uint) function

Article • 03/06/2023 • 2 minutes to read

Returns the red components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

Syntax

Syntax

```
TemplateType GatherRed(  
    in SamplerState S,  
    in float3      Location,  
    out uint       Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns **TRUE** if all values from the corresponding [Sample](#), [Gather](#), or [Load](#) operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns **FALSE**.

Return value

Type: `TemplateType`

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherRed methods](#)

[TextureCube](#)

TextureCube::Sample methods

Article • 03/09/2021 • 2 minutes to read

Samples a texture.

Overload list

Method	Description
Sample(S,float)	Samples a texture.
Sample(S,float,float)	Samples a texture with an optional value to clamp sample level-of-detail (LOD) values to.
Sample(S,float,float,uint)	Samples a texture with an optional value to clamp sample level-of-detail (LOD) values to, and returns status of the operation.

See also

[TextureCube](#)

[Texture-Object](#)

TextureCube::Sample(S,float,float) function

Article • 03/09/2021 • 2 minutes to read

Samples a texture with an optional value to clamp sample level-of-detail (LOD) values to.

Syntax

Syntax

```
DXGI_FORMAT Sample(  
    in SamplerState S,  
    in float        Location,  
    in float        Clamp  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[Sample methods](#)

[TextureCube](#)

TextureCube::Sample(S,float,float,uint) function

Article • 03/09/2021 • 2 minutes to read

Samples a texture with an optional value to clamp sample level-of-detail (LOD) values to, and returns status of the operation.

Syntax

Syntax

```
DXGI_FORMAT Sample(  
    in SamplerState S,  
    in float        Location,  
    in float        Clamp,  
    out uint       Status  
) ;
```

Parameters

S [in]

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

Clamp [in]

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Status [out]

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns TRUE if all values from the corresponding [Sample](#), [Gather](#), or [Load](#) operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns FALSE.

Return value

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[Sample methods](#)

[TextureCube](#)

TextureCube::SampleBias methods

Article • 03/09/2021 • 2 minutes to read

Samples a texture, after applying the bias value to the mipmap level.

Overload list

Method	Description
SampleBias(S,float,float)	Samples a texture, after applying the bias value to the mipmap level.
SampleBias(S,float,float,float)	Samples a texture, after applying the bias value to the mipmap level, with an optional value to clamp sample level-of-detail (LOD) values to.
SampleBias(S,float,float,float,uint)	Samples a texture, after applying the bias value to the mipmap level, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

See also

[TextureCube](#)

[Texture-Object](#)

SampleBias::SampleBias(S,float,float,floa t) function for TextureCube

Article • 04/02/2021 • 2 minutes to read

Samples a texture, after applying the bias value to the mipmap level, with an optional value to clamp sample level-of-detail (LOD) values to.

Syntax

Syntax

```
DXGI_FORMAT SampleBias(  
    in SamplerState S,  
    in float        Location,  
    in float        Bias,  
    in float        Clamp  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

Bias [in]

Type: **float**

The bias value, which is a floating-point number between 0.0 and 1.0 inclusive, is applied to a mip level before sampling.

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[SampleBias methods](#)

[TextureCube](#)

SampleBias::SampleBias(S,float,float,floa t,uint) function for TextureCube

Article • 04/02/2021 • 2 minutes to read

Samples a texture, after applying the bias value to the mipmap level, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

Syntax

syntax

```
DXGI_FORMAT SampleBias(  
    in SamplerState S,  
    in float        Location,  
    in float        Bias,  
    in float        Clamp,  
    out uint        Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3

Texture-Object Type	Parameter Type
TextureCubeArray	float4

Bias [in]

Type: **float**

The bias value, which is a floating-point number between 0.0 and 1.0 inclusive, is applied to a mip level before sampling.

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns **TRUE** if all values from the corresponding [Sample](#), [Gather](#), or [Load](#) operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns **FALSE**.

Return value

Type: [**DXGI_FORMAT**](#)

The texture format, which is one of the typed values listed in [**DXGI_FORMAT**](#).

See also

[SampleBias methods](#)

[TextureCube](#)

TextureCube::SampleCmp methods

Article • 03/09/2021 • 2 minutes to read

Samples a texture, using a comparison value to reject samples.

Overload list

Method	Description
SampleCmp(S,float,float)	Samples a texture, using a comparison value to reject samples.
SampleCmp(S,float,float,float)	Samples a texture, using a comparison value to reject samples, with an optional value to clamp sample level-of-detail (LOD) values to.
SampleCmp(S,float,float,float,uint)	Samples a texture, using a comparison value to reject samples, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

See also

[TextureCube](#)

[Texture-Object](#)

SampleCmp::SampleCmp(S,float,float,flo at) function for TextureCube

Article • 03/15/2021 • 2 minutes to read

Samples a texture, using a comparison value to reject samples, with an optional value to clamp sample level-of-detail (LOD) values to.

Syntax

Syntax

```
DXGI_FORMAT SampleCmp(  
    in SamplerState S,  
    in float        Location,  
    in float        CompareValue,  
    in float        Clamp  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

CompareValue [in]

Type: **float**

A floating-point value to use as a comparison value.

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[SampleCmp methods](#)

[TextureCube](#)

SampleCmp::SampleCmp(S,float,float,flo at,uint) function for TextureCube

Article • 03/15/2021 • 2 minutes to read

Samples a texture, using a comparison value to reject samples, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

Syntax

Syntax

```
DXGI_FORMAT SampleCmp(  
    in SamplerState S,  
    in float        Location,  
    in float        CompareValue,  
    in float        Clamp,  
    out uint        Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

CompareValue [in]

Type: **float**

A floating-point value to use as a comparison value.

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns **TRUE** if all values from the corresponding `Sample`, `Gather`, or `Load` operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns **FALSE**.

Return value

Type: [**DXGI_FORMAT**](#)

The texture format, which is one of the typed values listed in [**DXGI_FORMAT**](#).

See also

[SampleCmp methods](#)

[TextureCube](#)

TextureCube::SampleCmpLevelZero methods

Article • 03/09/2021 • 2 minutes to read

Samples a texture on mipmap level 0 only, using a comparison value to reject samples.

Overload list

Method	Description
SampleCmpLevelZero(S,float,float)	Samples a texture on mipmap level 0 only and compares the result to a comparison value.
SampleCmpLevelZero(S,float,float,uint)	Samples a texture on mipmap level 0 only and compares the result to a comparison value. Returns status about the operation.

See also

[TextureCube](#)

[Texture-Object](#)

SampleCmpLevelZero::SampleCmpLevelZero(S,float,float,uint) function for TextureCube

Article • 03/15/2021 • 2 minutes to read

Samples a texture on mipmap level 0 only and compares the result to a comparison value. Returns status about the operation.

Syntax

```
syntax

DXGI_FORMAT SampleCmpLevelZero(
    in SamplerState S,
    in float          Location,
    in float          CompareValue,
    out uint          Status
);
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3

Texture-Object Type	Parameter Type
TextureCubeArray	float4

CompareValue [in]

Type: **float**

A floating-point value to use as a comparison value.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns **FALSE**.

Return value

Type: [**DXGI_FORMAT**](#)

The texture format, which is one of the typed values listed in [**DXGI_FORMAT**](#).

See also

[SampleCmpLevelZero methods](#)

[TextureCube](#)

TextureCube::SampleGrad methods

Article • 03/09/2021 • 2 minutes to read

Samples a texture using a gradient to influence the way the sample location is calculated.

Overload list

Method	Description
SampleGrad(S,float,float,float)	Samples a texture, using a gradient to influence the way the sample location is calculated.
SampleGrad(S,float,float,float,float)	Samples a texture, using a gradient to influence the way the sample location is calculated, with an optional value to clamp sample level-of-detail (LOD) values to.
SampleGrad(S,float,float,float,float,uint)	Samples a texture, using a gradient to influence the way the sample location is calculated, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

See also

[TextureCube](#)

[Texture-Object](#)

SampleGrad::SampleGrad(S,float,float,float,float) function for TextureCube

Article • 03/15/2021 • 2 minutes to read

Samples a texture, using a gradient to influence the way the sample location is calculated, with an optional value to clamp sample level-of-detail (LOD) values to.

Syntax

Syntax

```
DXGI_FORMAT SampleGrad(  
    in SamplerState S,  
    in float        Location,  
    in float        DDX,  
    in float        DDY,  
    in float        Clamp  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

DDX [in]

Type: **float**

The rate of change of the surface geometry in the x direction. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	float
Texture2D, Texture2DArray	float2
Texture3D, TextureCube, TextureCubeArray	float3
Texture2DMS, Texture2DMSArray	not supported

DDY [in]

Type: **float**

The rate of change of the surface geometry in the y direction. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	float
Texture2D, Texture2DArray	float2
Texture3D, TextureCube, TextureCubeArray	float3
Texture2DMS, Texture2DMSArray	not supported

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[SampleGrad methods](#)

[TextureCube](#)

SampleGrad::SampleGrad(S,float,float,float,float,uint) function for TextureCube

Article • 03/15/2021 • 2 minutes to read

Samples a texture, using a gradient to influence the way the sample location is calculated, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

Syntax

syntax

```
DXGI_FORMAT SampleGrad(  
    in SamplerState S,  
    in float        Location,  
    in float        DDX,  
    in float        DDY,  
    in float        Clamp,  
    out uint       Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3

Texture-Object Type	Parameter Type
TextureCubeArray	float4

DDX [in]

Type: **float**

The rate of change of the surface geometry in the x direction. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	float
Texture2D, Texture2DArray	float2
Texture3D, TextureCube, TextureCubeArray	float3
Texture2DMS, Texture2DMSArray	not supported

DDY [in]

Type: **float**

The rate of change of the surface geometry in the y direction. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	float
Texture2D, Texture2DArray	float2
Texture3D, TextureCube, TextureCubeArray	float3
Texture2DMS, Texture2DMSArray	not supported

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Status [out]

Type: `uint`

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns `TRUE` if all values from the corresponding `Sample`, `Gather`, or `Load` operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns `FALSE`.

Return value

Type: [`DXGI_FORMAT`](#)

The texture format, which is one of the typed values listed in [`DXGI_FORMAT`](#).

See also

[SampleGrad methods](#)

[TextureCube](#)

TextureCube::SampleLevel methods

Article • 03/09/2021 • 2 minutes to read

Samples a texture on the specified mipmap level.

Overload list

Method	Description
SampleLevel(S,float,float)	Samples a texture on the specified mipmap level.
SampleLevel(S,float,float,uint)	Samples a texture on the specified mipmap level and returns status about the operation.

See also

[TextureCube](#)

[Texture-Object](#)

SampleLevel::SampleLevel(S,float,float,uint) function for TextureCube

Article • 03/15/2021 • 2 minutes to read

Samples a texture on the specified mipmap level and returns status about the operation.

Syntax

syntax

```
DXGI_FORMAT SampleLevel(  
    in SamplerState S,  
    in float        Location,  
    in float        LOD,  
    out uint        Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

LOD [in]

Type: **float**

[in] A number that specifies the mipmap level. If the value is ≤ 0 , mipmap level 0 (biggest map) is used. The fractional value (if supplied) is used to interpolate between two mipmap levels.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns **TRUE** if all values from the corresponding [Sample](#), [Gather](#), or [Load](#) operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns **FALSE**.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[SampleLevel methods](#)

[TextureCube](#)

TextureCubeArray object

Article • 02/12/2021 • 2 minutes to read

TextureCubeArray type ([as it exists in Shader Model 4](#)) plus resource variables. This texture object supports these methods in addition to the methods in Shader Model 4.

- [Methods](#)

Methods

The TextureCubeArray object has these methods.

Method	Description
Gather	Returns the four texel values that would be used in a bi-linear filtering operation.
GatherAlpha	Returns the alpha components of the four texel values that would be used in a bi-linear filtering operation.
GatherBlue	Returns the blue components of the four texel values that would be used in a bi-linear filtering operation.
GatherCmp	For four texel values that would be used in a bi-linear filtering operation, returns their comparison against a compare value.
GatherCmpAlpha	For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their alpha component against a compare value.
GatherCmpBlue	For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their blue component against a compare value.
GatherCmpGreen	For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their green component against a compare value.
GatherCmpRed	For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their red component against a compare value.
GatherGreen	Returns the green components of the four texel values that would be used in a bi-linear filtering operation.
GatherRed	Returns the red components of the four texel values that would be used in a bi-linear filtering operation.
Sample	Samples a texture.
SampleBias	Samples a texture, after applying the bias value to the mipmap level.

Method	Description
SampleCmp	Samples a texture, using a comparison value to reject samples.
SampleCmpLevelZero	Samples a texture (mipmap level 0 only), using a comparison value to reject samples.
SampleGrad	Samples a texture using a gradient to influence the way the sample location is calculated.
SampleLevel	Samples a texture on the specified mipmap level.

Remarks

Minimum Shader Model

This object is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This object is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Shader Model 5 Objects](#)

TextureCubeArray::Gather methods

Article • 03/09/2021 • 2 minutes to read

Returns the red components of four texel values that would be used in a bi-linear filtering operation.

See the documentation on [gather4](#) for more information describing the underlying DXBC instruction.

Overload list

Method	Description
Gather(S,float)	Returns the red components of four texel values (red component only) that would be used in a bi-linear filtering operation.
Gather(S,float,uint)	Returns the red components of four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

See also

[TextureCubeArray](#)

TextureCubeArray::Gather(S,float,uint) function

Article • 03/06/2023 • 2 minutes to read

Returns the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

Syntax

Syntax

```
TemplateType Gather(  
    in SamplerState S,  
    in float4      Location,  
    out uint       Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns **TRUE** if all values from the corresponding [Sample](#), [Gather](#), or [Load](#) operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns **FALSE**.

Return value

Type: `TemplateType`

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Gather methods](#)

[`TextureCubeArray`](#)

TextureCubeArray::GatherAlpha methods

Article • 03/09/2021 • 2 minutes to read

Returns the alpha components of the four texel values that would be used in a bi-linear filtering operation.

Overload list

Method	Description
GatherAlpha(S,float,uint)	Returns the alpha components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

See also

[TextureCubeArray](#)

TextureCubeArray::GatherAlpha(S,float,uint) function

Article • 03/06/2023 • 2 minutes to read

Returns the alpha components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

Syntax

Syntax

```
TemplateType GatherAlpha(  
    in SamplerState S,  
    in float4      Location,  
    out uint       Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns **FALSE**.

Return value

Type: [TemplateType](#)

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherAlpha methods](#)

[TextureCubeArray](#)

TextureCubeArray::GatherBlue methods

Article • 03/09/2021 • 2 minutes to read

Returns the blue components of the four texel values that would be used in a bi-linear filtering operation.

Overload list

Method	Description
GatherBlue(S,float,uint)	Returns the blue components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

See also

[TextureCubeArray](#)

TextureCubeArray::GatherBlue(S,float,uint) function

Article • 03/06/2023 • 2 minutes to read

Returns the blue components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

Syntax

Syntax

```
TemplateType GatherBlue(  
    in SamplerState S,  
    in float4      Location,  
    out uint       Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns **TRUE** if all values from the corresponding [Sample](#), [Gather](#), or [Load](#) operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns **FALSE**.

Return value

Type: `TemplateType`

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherBlue methods](#)

[TextureCubeArray](#)

TextureCubeArray::GatherCmp methods

Article • 03/09/2021 • 2 minutes to read

For four texel values of a [TextureCubeArray](#) that would be used in a bi-linear filtering operation, returns their comparison against a compare value.

See the documentation on [gather4_c](#) for more information describing the underlying DXBC instruction.

Overload list

Method	Description
GatherCmp(S,float,float,uint)	Samples and compares a texture and returns all four components along with status about the operation.

See also

[TextureCubeArray](#)

TextureCubeArray::GatherCmp(S,float,float,uint) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns their comparison against a compare value along with tile-mapping status.

Syntax

Syntax

```
TemplateType GatherCmp(  
    in SamplerState S,  
    in float        Location,  
    in float        CompareValue,  
    out uint        Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

CompareValue [in]

Type: **float**

A value to compare each against each sampled value.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns **FALSE**.

Return value

Type: [TemplateType](#)

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmp methods](#)

[TextureCubeArray](#)

TextureCubeArray::GatherCmpAlpha methods

Article • 03/09/2021 • 2 minutes to read

Samples and compares a texture and returns the alpha component.

Overload list

Method	Description
GatherCmpAlpha(S,float,float,uint)	Samples and compares a texture and returns the alpha component along with status about the operation.

See also

[TextureCubeArray](#)

TextureCubeArray::GatherCmpAlpha(S,float,uint) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their alpha component against a compare value along with tile-mapping status.

Syntax

syntax

```
TemplateType GatherCmpAlpha(  
    in SamplerState S,  
    in float      Location,  
    in float      CompareValue,  
    out uint      Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

CompareValue [in]

Type: **float**

A value to compare each against each sampled value.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns **FALSE**.

Return value

Type: [TemplateType](#)

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmpAlpha methods](#)

[TextureCubeArray](#)

TextureCubeArray::GatherCmpBlue methods

Article • 03/09/2021 • 2 minutes to read

Samples and compares a texture and returns the blue component.

Overload list

Method	Description
GatherCmpBlue(S,float,float,uint)	Samples and compares a texture and returns the blue component along with status about the operation.

See also

[TextureCubeArray](#)

TextureCubeArray::GatherCmpBlue(S,float,uint) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their blue component against a compare value along with tile-mapping status.

Syntax

syntax

```
TemplateType GatherCmpBlue(  
    in SamplerState S,  
    in float      Location,  
    in float      CompareValue,  
    out uint      Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

CompareValue [in]

Type: **float**

A value to compare each against each sampled value.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns **FALSE**.

Return value

Type: [TemplateType](#)

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmpBlue methods](#)

[TextureCubeArray](#)

TextureCubeArray::GatherCmpGreen methods

Article • 03/09/2021 • 2 minutes to read

Samples and compares a texture and returns the green component.

Overload list

Method	Description
GatherCmpGreen(S,float,float,uint)	Samples and compares a texture and returns the green component along with status about the operation.

See also

[TextureCubeArray](#)

TextureCubeArray::GatherCmpGreen(S,float,uint) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their green component against a compare value along with tile-mapping status.

Syntax

syntax

```
TemplateType GatherCmpGreen(  
    in SamplerState S,  
    in float      Location,  
    in float      CompareValue,  
    out uint      Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

CompareValue [in]

Type: **float**

A value to compare each against each sampled value.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns **FALSE**.

Return value

Type: [TemplateType](#)

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmpGreen methods](#)

[TextureCubeArray](#)

TextureCubeArray::GatherCmpRed methods

Article • 03/09/2021 • 2 minutes to read

Samples and compares a texture and returns the red component.

Overload list

Method	Description
GatherCmpRed(S,float,float,uint)	Samples and compares a texture and returns the red component along with status about the operation.

See also

[TextureCubeArray](#)

TextureCubeArray::GatherCmpRed(S,float,uint) function

Article • 03/09/2021 • 2 minutes to read

For four texel values that would be used in a bi-linear filtering operation, returns a comparison of their red component against a compare value along with tile-mapping status.

Syntax

syntax

```
TemplateType GatherCmpRed(  
    in SamplerState S,  
    in float        Location,  
    in float        CompareValue,  
    out uint        Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

CompareValue [in]

Type: **float**

A value to compare each against each sampled value.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns **TRUE** if all values from the corresponding **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns **FALSE**.

Return value

Type: [TemplateType](#)

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherCmpRed methods](#)

[TextureCubeArray](#)

TextureCubeArray::GatherGreen methods

Article • 03/09/2021 • 2 minutes to read

Samples a texture and returns the green component.

Overload list

Method	Description
GatherGreen(S,float,uint)	Returns the green components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

See also

[TextureCubeArray](#)

TextureCubeArray::GatherGreen(S,float,uint) function

Article • 03/06/2023 • 2 minutes to read

Returns the green components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

Syntax

Syntax

```
TemplateType GatherGreen(  
    in SamplerState S,  
    in float4      Location,  
    out uint       Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns **TRUE** if all values from the corresponding [Sample](#), [Gather](#), or [Load](#) operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns **FALSE**.

Return value

Type: [TemplateType](#)

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherGreen methods](#)

[TextureCubeArray](#)

TextureCubeArray::GatherRed methods

Article • 03/09/2021 • 2 minutes to read

Returns the red components of the four texel values that would be used in a bi-linear filtering operation.

Overload list

Method	Description
GatherRed(S,float,uint)	Returns the red components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

See also

[TextureCubeArray](#)

TextureCubeArray::GatherRed(S,float,uint) function

Article • 03/06/2023 • 2 minutes to read

Returns the red components of the four texel values that would be used in a bi-linear filtering operation, along with tile-mapping status.

Syntax

Syntax

```
TemplateType GatherRed(  
    in SamplerState S,  
    in float4      Location,  
    out uint       Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

The zero-based sampler index.

Location [in]

Type: **float**

The sample coordinates (u,v).

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns **TRUE** if all values from the corresponding [Sample](#), [Gather](#), or [Load](#) operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns **FALSE**.

Return value

Type: `TemplateType`

A four-component value whose type is the same as the template type.

Remarks

The texture samples can be used for bilinear interpolation.

This function is supported for the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[GatherRed methods](#)

[TextureCubeArray](#)

TextureCubeArray::Sample methods

Article • 03/09/2021 • 2 minutes to read

Samples a texture.

Overload list

Method	Description
Sample(S,float)	Samples a texture.
Sample(S,float,float)	Samples a texture with an optional value to clamp sample level-of-detail (LOD) values to.
Sample(S,float,float,uint)	Samples a texture with an optional value to clamp sample level-of-detail (LOD) values to, and returns status of the operation.

See also

[TextureCubeArray](#)

[Texture-Object](#)

TextureCubeArray::Sample(S,float,float) function

Article • 03/09/2021 • 2 minutes to read

Samples a texture with an optional value to clamp sample level-of-detail (LOD) values to.

Syntax

Syntax

```
DXGI_FORMAT Sample(  
    in SamplerState S,  
    in float        Location,  
    in float        Clamp  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[Sample methods](#)

[TextureCubeArray](#)

TextureCubeArray::Sample(S,float,float,uint) function

Article • 03/09/2021 • 2 minutes to read

Samples a texture with an optional value to clamp sample level-of-detail (LOD) values to, and returns status of the operation.

Syntax

Syntax

```
DXGI_FORMAT Sample(  
    in SamplerState S,  
    in float        Location,  
    in float        Clamp,  
    out uint        Status  
) ;
```

Parameters

S [in]

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

Clamp [in]

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Status [out]

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns TRUE if all values from the corresponding [Sample](#), [Gather](#), or [Load](#) operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns FALSE.

Return value

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[Sample methods](#)

[TextureCubeArray](#)

TextureCubeArray::SampleBias methods

Article • 03/09/2021 • 2 minutes to read

Samples a texture, after applying the bias value to the mipmap level.

Overload list

Method	Description
SampleBias(S,float,float)	Samples a texture, after applying the bias value to the mipmap level.
SampleBias(S,float,float,float)	Samples a texture, after applying the bias value to the mipmap level, with an optional value to clamp sample level-of-detail (LOD) values to.
SampleBias(S,float,float,float,uint)	Samples a texture, after applying the bias value to the mipmap level, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

See also

[TextureCubeArray](#)

[Texture-Object](#)

SampleBias::SampleBias(S,float,float,floa t) function for TextureCubeArray

Article • 04/02/2021 • 2 minutes to read

Samples a texture, after applying the bias value to the mipmap level, with an optional value to clamp sample level-of-detail (LOD) values to.

Syntax

Syntax

```
DXGI_FORMAT SampleBias(  
    in SamplerState S,  
    in float        Location,  
    in float        Bias,  
    in float        Clamp  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

Bias [in]

Type: **float**

The bias value, which is a floating-point number between 0.0 and 1.0 inclusive, is applied to a mip level before sampling.

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[SampleBias methods](#)

[TextureCubeArray](#)

SampleBias::SampleBias(S,float,float,floa t,uint) function for TextureCubeArray

Article • 04/02/2021 • 2 minutes to read

Samples a texture, after applying the bias value to the mipmap level, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

Syntax

syntax

```
DXGI_FORMAT SampleBias(  
    in SamplerState S,  
    in float        Location,  
    in float        Bias,  
    in float        Clamp,  
    out uint        Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3

Texture-Object Type	Parameter Type
TextureCubeArray	float4

Bias [in]

Type: **float**

The bias value, which is a floating-point number between 0.0 and 1.0 inclusive, is applied to a mip level before sampling.

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns **TRUE** if all values from the corresponding [Sample](#), [Gather](#), or [Load](#) operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns **FALSE**.

Return value

Type: [**DXGI_FORMAT**](#)

The texture format, which is one of the typed values listed in [**DXGI_FORMAT**](#).

See also

[SampleBias methods](#)

[TextureCubeArray](#)

TextureCubeArray::SampleCmp methods

Article • 03/09/2021 • 2 minutes to read

Samples a texture, using a comparison value to reject samples.

Overload list

Method	Description
SampleCmp(S,float,float)	Samples a texture, using a comparison value to reject samples.
SampleCmp(S,float,float,float)	Samples a texture, using a comparison value to reject samples, with an optional value to clamp sample level-of-detail (LOD) values to.
SampleCmp(S,float,float,float,uint)	Samples a texture, using a comparison value to reject samples, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

See also

[TextureCubeArray](#)

[Texture-Object](#)

SampleCmp::SampleCmp(S,float,float,flo at) function for TextureCubeArray

Article • 03/15/2021 • 2 minutes to read

Samples a texture, using a comparison value to reject samples, with an optional value to clamp sample level-of-detail (LOD) values to.

Syntax

Syntax

```
DXGI_FORMAT SampleCmp(  
    in SamplerState S,  
    in float        Location,  
    in float        CompareValue,  
    in float        Clamp  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

CompareValue [in]

Type: **float**

A floating-point value to use as a comparison value.

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[SampleCmp methods](#)

[TextureCubeArray](#)

SampleCmp::SampleCmp(S,float,float,flo at,uint) function for TextureCubeArray

Article • 03/15/2021 • 2 minutes to read

Samples a texture, using a comparison value to reject samples, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

Syntax

Syntax

```
DXGI_FORMAT SampleCmp(  
    in SamplerState S,  
    in float          Location,  
    in float          CompareValue,  
    in float          Clamp,  
    out uint          Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

CompareValue [in]

Type: **float**

A floating-point value to use as a comparison value.

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns **TRUE** if all values from the corresponding `Sample`, `Gather`, or `Load` operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns **FALSE**.

Return value

Type: [**DXGI_FORMAT**](#)

The texture format, which is one of the typed values listed in [**DXGI_FORMAT**](#).

See also

[SampleCmp methods](#)

[TextureCubeArray](#)

TextureCubeArray::SampleCmpLevelZero methods

Article • 03/09/2021 • 2 minutes to read

Samples a texture on mipmap level 0 only, using a comparison value to reject samples.

Overload list

Method	Description
SampleCmpLevelZero(S,float,float)	Samples a texture on mipmap level 0 only and compares the result to a comparison value.
SampleCmpLevelZero(S,float,float,uint)	Samples a texture on mipmap level 0 only and compares the result to a comparison value. Returns status about the operation.

See also

[TextureCubeArray](#)

[Texture-Object](#)

SampleCmpLevelZero::SampleCmpLevelZero(S,float,float,uint) function for TextureCubeArray

Article • 03/15/2021 • 2 minutes to read

Samples a texture on mipmap level 0 only and compares the result to a comparison value. Returns status about the operation.

Syntax

```
syntax

DXGI_FORMAT SampleCmpLevelZero(
    in SamplerState S,
    in float          Location,
    in float          CompareValue,
    out uint          Status
);
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3

Texture-Object Type	Parameter Type
TextureCubeArray	float4

CompareValue [in]

Type: **float**

A floating-point value to use as a comparison value.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns **TRUE** if all values from the corresponding `Sample`, `Gather`, or `Load` operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns **FALSE**.

Return value

Type: [**DXGI_FORMAT**](#)

The texture format, which is one of the typed values listed in [**DXGI_FORMAT**](#).

See also

[SampleCmpLevelZero methods](#)

[TextureCubeArray](#)

TextureCubeArray::SampleGrad methods

Article • 03/09/2021 • 2 minutes to read

Samples a texture using a gradient to influence the way the sample location is calculated.

Overload list

Method	Description
SampleGrad(S,float,float,float)	Samples a texture, using a gradient to influence the way the sample location is calculated.
SampleGrad(S,float,float,float,float)	Samples a texture, using a gradient to influence the way the sample location is calculated, with an optional value to clamp sample level-of-detail (LOD) values to.
SampleGrad(S,float,float,float,float,uint)	Samples a texture, using a gradient to influence the way the sample location is calculated, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

See also

[TextureCubeArray](#)

[Texture-Object](#)

SampleGrad::SampleGrad(S,float,float,float,float) function for TextureCubeArray

Article • 03/15/2021 • 2 minutes to read

Samples a texture, using a gradient to influence the way the sample location is calculated, with an optional value to clamp sample level-of-detail (LOD) values to.

Syntax

Syntax

```
DXGI_FORMAT SampleGrad(  
    in SamplerState S,  
    in float        Location,  
    in float        DDX,  
    in float        DDY,  
    in float        Clamp  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

DDX [in]

Type: **float**

The rate of change of the surface geometry in the x direction. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	float
Texture2D, Texture2DArray	float2
Texture3D, TextureCube, TextureCubeArray	float3
Texture2DMS, Texture2DMSArray	not supported

DDY [in]

Type: **float**

The rate of change of the surface geometry in the y direction. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	float
Texture2D, Texture2DArray	float2
Texture3D, TextureCube, TextureCubeArray	float3
Texture2DMS, Texture2DMSArray	not supported

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[SampleGrad methods](#)

[TextureCubeArray](#)

SampleGrad::SampleGrad(S,float,float,fl oat,float,uint) function for TextureCubeArray

Article • 03/15/2021 • 2 minutes to read

Samples a texture, using a gradient to influence the way the sample location is calculated, with an optional value to clamp sample level-of-detail (LOD) values to. Returns status about the operation.

Syntax

syntax

```
DXGI_FORMAT SampleGrad(  
    in SamplerState S,  
    in float        Location,  
    in float        DDX,  
    in float        DDY,  
    in float        Clamp,  
    out uint       Status  
)
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2

Texture-Object Type	Parameter Type
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

DDX [in]

Type: **float**

The rate of change of the surface geometry in the x direction. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	float
Texture2D, Texture2DArray	float2
Texture3D, TextureCube, TextureCubeArray	float3
Texture2DMS, Texture2DMSArray	not supported

DDY [in]

Type: **float**

The rate of change of the surface geometry in the y direction. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D, Texture1DArray	float
Texture2D, Texture2DArray	float2
Texture3D, TextureCube, TextureCubeArray	float3
Texture2DMS, Texture2DMSArray	not supported

Clamp [in]

Type: **float**

An optional value to clamp sample LOD values to. For example, if you pass 2.0f for the clamp value, you ensure that no individual sample accesses a mip level less than 2.0f.

Status [out]

Type: `uint`

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. `CheckAccessFullyMapped` returns `TRUE` if all values from the corresponding `Sample`, `Gather`, or `Load` operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, `CheckAccessFullyMapped` returns `FALSE`.

Return value

Type: [`DXGI_FORMAT`](#)

The texture format, which is one of the typed values listed in [`DXGI_FORMAT`](#).

See also

[SampleGrad methods](#)

[TextureCubeArray](#)

TextureCubeArray::SampleLevel methods

Article • 03/09/2021 • 2 minutes to read

Samples a texture on the specified mipmap level.

Overload list

Method	Description
SampleLevel(S,float,float)	Samples a texture on the specified mipmap level.
SampleLevel(S,float,float,uint)	Samples a texture on the specified mipmap level and returns status about the operation.

See also

[TextureCubeArray](#)

[Texture-Object](#)

SampleLevel::SampleLevel(S,float,float,uint) function

Article • 03/09/2021 • 2 minutes to read

Samples a texture on the specified mipmap level and returns status about the operation.

Syntax

syntax

```
DXGI_FORMAT SampleLevel(  
    in SamplerState S,  
    in float        Location,  
    in float        LOD,  
    out uint        Status  
) ;
```

Parameters

S [in]

Type: **SamplerState**

A [Sampler state](#). This is an object declared in an effect file that contains state assignments.

Location [in]

Type: **float**

The texture coordinates. The argument type is dependent on the texture-object type.

Texture-Object Type	Parameter Type
Texture1D	float
Texture1DArray, Texture2D	float2
Texture2DArray, Texture3D, TextureCube	float3
TextureCubeArray	float4

LOD [in]

Type: **float**

[in] A number that specifies the mipmap level. If the value is ≤ 0 , mipmap level 0 (biggest map) is used. The fractional value (if supplied) is used to interpolate between two mipmap levels.

Status [out]

Type: **uint**

The status of the operation. You can't access the status directly; instead, pass the status to the [CheckAccessFullyMapped](#) intrinsic function. [CheckAccessFullyMapped](#) returns **TRUE** if all values from the corresponding [Sample](#), [Gather](#), or [Load](#) operation accessed mapped tiles in a [tiled resource](#). If any values were taken from an unmapped tile, [CheckAccessFullyMapped](#) returns **FALSE**.

Return value

Type: [DXGI_FORMAT](#)

The texture format, which is one of the typed values listed in [DXGI_FORMAT](#).

See also

[SampleLevel methods](#)

[TextureCubeArray](#)

Shader Model 5 System Values

Article • 11/20/2019 • 2 minutes to read

Shader Model 5 implements the [system values](#) from Shader Model 4 and earlier, as well as the following new system values:

- [SV_DispatchThreadID](#)
- [SV_DomainLocation](#)
- [SV_GroupID](#)
- [SV_GroupIndex](#)
- [SV_GroupThreadID](#)
- [SV_GSInstanceID](#)
- [SV_InsideTessFactor](#)
- [SV_OutputControlPointID](#)
- [SV_TessFactor](#)

Related topics

[Shader Model 5](#)

Registers - vs_5_0

Article • 08/23/2019 • 2 minutes to read

The following input and output registers are implemented in the vertex shader version 5_0.

Input Registers

Register Type	Count	R/W	Dimension	Indexable by r#	Defaults	Requires DCL
32-bit Temp (r#)	4096(r#+x#[n])	R/W	4	No	None	Yes
32-bit indexable Temp Array (x#[n])	4096(r#+x#[n])	R/W	4	Yes	None	Yes
32-bit input (v#)	32	R	4	Yes	None	Yes
Element in an input resource (t#)	128	R	1	No	None	Yes
Sampler (s#)	16	R	1	No	None	Yes
ConstantBuffer reference (cb#[index])	15	R	4	Yes(contents)	None	Yes
Immediate ConstantBuffer reference (icb[index])	1	R	4	Yes(contents)	None	Yes

Output Registers

Register Type	Count	R/W	Dimension	Indexable by r#	Defaults	Requires DCL
NULL (discard result, useful for operations with multiple results)	N/A	W	N/A	N/A	N/A	No
32-bit output Vertex Data Element (o#)	32	W	4	N/A	N/A	Yes

Related topics

[Shader Model 5](#)

Registers - ps_5_0

Article • 08/23/2019 • 2 minutes to read

The following input and output registers are implemented in the pixel shader version 5_0.

Input Registers

Register Type	Count	R/W	Dimension	Indexable by r#	Defaults	Requires DCL
32-bit Temp (r#)	4096(r#+x#[n])	R/W	4	No	None	Yes
32-bit Indexable Temp Array (x#[n])	4096(r#+x#[n])	R/W	4	Yes	None	Yes
32-bit Input Attribute (v#)	32	R	4	Yes	None	Yes
Element in an input resource (t#)	128	R	1	No	None	Yes
Sampler (s#)	16	R	1	No	None	Yes
ConstantBuffer reference (cb#[index])	15	R	4	Yes(contents)	None	Yes
Immediate ConstantBuffer reference (icb[index])	1	R	4	Yes(contents)	None	Yes

Output Registers

Register Type	Count	R/W	Dimension	Indexable by r#	Defaults	Requires DCL
NULL (discard result, useful for operations with multiple results)	N/A	W	N/A	N/A	N/A	No
32-bit output Element (o#)	8	W	4	N/A	N/A	No

Register Type	Count	R/W	Dimension	Indexable by r#	Defaults	Requires DCL
Unordered Access View (u#)	8 - # of rendertargets	R/W	D3D11_PS_CS_UAV_REGISTER_COMPONENTS	No	No	Yes
32-bit float output depth (oDepth)	1	W	1	N/A	N/A	Yes
32-bit UINT output sample mask (oMask)	1	W	1	N/A	N/A	Yes

Related topics

[Shader Model 5](#)

Registers - gs_5_0

Article • 08/23/2019 • 2 minutes to read

The following input and output registers are implemented in the geometry shader version 5_0.

Input Registers

Register Type	Count	R/W	Dimension	Indexable by r#	Defaults	Requires DCL
32-bit Temp (r#)	4096(r#+x#[n])	R/W	4	No	None	Yes
32-bit Indexable Temp Array (x#[n])	4096(r#+x#[n])	R/W	4	Yes	None	Yes
32-bit Input (v[vertex][element])	32	R	4(comp)*32(vert)	Yes	None	Yes
32-bit Input Primitive ID (vPrim)	1	R	1	No	None	Yes
32-bit Input Instance ID (vInstanceID)	1	R	1	No	None	Yes
Element in an input resource (t#)	128	R	1	No	None	Yes
Sampler (s#)	16	R	1	No	None	Yes
ConstantBuffer reference (cb#[index])	15	R	4	Yes(contents)	None	Yes
Immediate ConstantBuffer reference (icb[index])	1	R	4	Yes(contents)	None	Yes

Output Registers

Register Type	Count	R/W	Dimension	Indexable by r#	Defaults	Requires DCL
NULL (discard result, useful for ops with multiple results)	N/A	W	N/A	N/A	N/A	No
32-bit output Vertex Data Element (o#)	32	W	N/A	N/A	4	Yes

Related topics

[Shader Model 5](#)

Registers - hs_5_0

Article • 03/05/2021 • 6 minutes to read

A hull shader consists of three distinct phases: control point phase, fork phase, and join phase. Each phase has its own sets of input and output registers.

Control Point Phase

hs_control_point_phase is a shader program with the following register model.

Input Registers

Register Type	Count	R/W	Dimension	Indexable by r#	Defaults	Requires DCL
32-bit Temp (r#)	4096(r#+x#[n])	R/W	4	No	None	Yes
32-bit indexable Temp Array (x#[n])	4096(r#+x#[n])	R/W	4	Yes	None	Yes
32-bit Input (v[vertex] [element])	32(element)*32(vert)	R	4	Yes	None	Yes
32-bit UINT Input vOutputControlPointID(23.7)	1	R	1	No	None	Yes
32-bit UINT Input PrimitiveID (vPrim)	1	R	1	No	N/A	Yes
Element in an input resource (t#)	128	R	128	Yes	None	Yes
Sampler (s#)	16	R	1	Yes	None	Yes
ConstantBuffer reference (cb#[index])	15	R	4	Yes	None	Yes
Immediate ConstantBuffer reference (icb#[index])	1	R	4	Yes (contents)	None	Yes

Output Registers

Register Type	Count	R/W	Dimension	Indexable by r#	Defaults	Requires DCL
32-bit output Vertex Data Element (o#)	32	W	4	Yes	None	Yes

Each hull shader control point phase output register is up to a 4-vector, of which up to 32 registers can be declared. There are also from 1 to 32 output control points declared, which scales amount of storage required. Let us refer to the maximum allowable aggregate number of scalars across all hull shader control point phase output as `#cp_output_max`.

`#cp_output_max = 3968` scalars.

This limit is based on a design point for certain hardware of 4096×32 -bit storage here. The amount for control point output is $3968 = 4096 - 128$, which is $32(\text{control points})^4(\text{components})^32(\text{elements}) - 4(\text{components})^32(\text{elements})$. The subtraction reserves 128 scalars (one control point) worth of space dedicated to the hull shader phase 2 and 3. The choice of reserving 128 scalars for patch constants -- rather than allowing the amount to be simply whatever of the 4096 scalars of storage is unused by output control points -- accommodates the limits of another particular hardware design. The control point phase can declare 32 output control points, but they just can't be fully 32 elements with 4 components each, because the total storage would be too high.

Fork Phase

The following registers are visible in the `hs_fork_phase` model.

The input resources (`t#`), samplers (`s#`), constant buffers (`cb#`) and immediate constant buffer (`icb`) below are all shared state with all other hull shader phases. That is, from the API/DDI point of view, the hull shader has a single set of input resource state for all phases. This goes with the fact that from the API/DDI point of view, the hull shader is a single atomic shader; the phases within it are implementation details.

Input Registers

Register Type	Count	R/W	Dimension	Indexable by r#	Defaults	Requires DCL
32-bit Temp (<code>r#</code>)	$4096(r\#+x#[n])$	R/W	4	No	None	Yes
32-bit indexable Temp Array (<code>x#[n]</code>)	$4096(r\#+x#[n])$	R/W	4	Yes	None	Yes
32-bit Input Control Points (<code>vicp[vertex][element] (pre- Control Point Phase)</code>)	32 See note below	R	$4(\text{component})^32(\text{element})^32(\text{vert})$	Yes	None	Yes

Register Type	Count	R/W	Dimension	Indexable by r#	Defaults	Requires DCL
32-bit Output Control Points (vocp[vertex] [element]) (post- Control Point Phase)	32 See note below	R	4(component)*32(element)*32(vert)	Yes	None	Yes
32-bit UINT Input PrimitiveID (vPrim)	1	R	1	No	N/A	Yes
32-bit UINT Input ForkInstanceID(23.8) (vForkInstanceID)	1	R	1	No	N/A	Yes
Element in an input resource (t#)	128	R	128	Yes	None	Yes
Sampler (s#)	16	R	1	Yes	None	Yes
ConstantBuffer reference (cb# [index])	15	R	4	Yes	None	Yes
Immediate ConstantBuffer reference (icb[index])	1	R	4	Yes (contents)	None	Yes

⚠ Note

The hull shader fork phase's input control point register (vicp) declarations must be any subset, along the [element] axis, of the hull shader control point input (pre-control point phase). Similarly the declarations for inputting the output control points (vocp) must be any subset, along the [element] axis, of the hull shader output control points (post-control point phase).

Along the [vertex] axis, the number of control points to be read for each of the vicp and vocp must similarly be a subset of the hull shader input control point count and hull shader output control point count, respectively. For example, if the vertex axis of the vocp registers are declared with n vertices, that makes the control point phase's output control points [0..n-1] available as read only input to the fork phase.

Output Registers

Register	Count	R/W	Dimension	Indexable by r#	Defaults	Requires DCL
----------	-------	-----	-----------	-----------------	----------	--------------

Register	Count	R/W	Dimension	Indexable by r#	Defaults	Requires DCL
32-bit output Patch Constant Data Element (o#)	32 See note 1 below	W	4	Yes	None	Yes

① Note

The hull shader fork and join phase outputs are a shared set of 4 4-vector registers. The outputs of each fork or join phase program cannot overlap with each other. System-interpreted values such as TessFactors come out of this space.

Join Phase

The following registers are visible in the hs_join_phase model. There are three sets of input registers: control point phase input control points (vicp), vopc control point phase output control points (vopc), and patch constants (vcpc). vpc are the aggregate output of all the hull shader fork phase programs. The hull shader join phase output o# registers are in the same register space as the hulll shader fork phase outputs.

The input resources (t#), samplers (s#), constant buffers (cb#) and immediate constant buffer (icb) below are all shared state with all other hull shader phases. That is, from the API/DDI point of view, the hull shader has a single set of input resource state for all phases. This goes with the fact that from the API/DDI point of view, the hull shader is a single atomic shader; the phases within it are implementation details.

Input Registers

Register Type	Count	R/W	Dimension	Indexable by r#	Defaults	Requires DCL
32-bit Temp (r#)	4096(r#+x#[n])	R/W	4	No	None	Yes
32-bit indexable Temp Array (x#[n])	4096(r#+x#[n])	R/W	4	Yes	None	Yes
32-bit Input Control Points (vicp[vertex][element]) (pre- Control Point Phase)	32 See Note 1 below	R	4(component)*32(element)*32(vert)	Yes	None	Yes

Register Type	Count	R/W	Dimension	Indexable by r#	Defaults	Requires DCL
32-bit Output Control Points (vocp[vertex] [element]) (post-Control Point Phase)	32 See Note 1 below	R	4(component)*32(element)*32(vert)	Yes	None	Yes
32-bit Input (vpc[element]) (Patch Constant Data)	32 See Note 2 below	R	4	Yes	None	Yes
32-bit UINT Input PrimitiveID (vPrim)	1	R	1	No	N/A	Yes
32-bit UINT Input JoinInstanceID (vJoinInstanceID)	1	R	1	No	N/A	Yes
Element in an input resource (t#)	128	R	128	Yes	None	Yes
Sampler (s#)	16	R	1	Yes	None	Yes
ConstantBuffer reference (cb#[index])	15	R	4	Yes	None	Yes
Immediate ConstantBuffer reference (icb[index])	1	R	4	Yes (contents)	None	Yes

Note 1: The hull shader join phase's input control point register (vicp) declarations must be any subset, along the [element] axis, of the hull shader control point input (pre-control point phase). Similarly the declarations for inputting the output control points (vocp) must be any subset, along the [element] axis, of the hull shader output control points (post-control point phase).

Along the [vertex] axis, the number of control points to be read for each of the vicp and vocp must similarly be a subset of the hull shader input control point count and hull shader output control point count, respectively. For example, if the vertex axis of the vocp registers are declared with n vertices, that makes the control point phase's output control points [0..n-1] available as read only input to the join phase.

Note 2: In addition to control point input, the hull shader join phase also sees as input the patch constant data computed by the hull shader fork phase program(s). This shows up at the hull shader fork phase as the vpc# registers. The hull shader join phase's input vpc# registers share the same register space as the hull shader fork phase output o# registers. The declarations of the o# registers must not overlap with any hull shader fork phase program o# output declaration; the hull shader join phase is adding to the aggregate patch constant data output for the hull shader.

Output Registers

Register Type	Count	R/W	Dimension	Indexable by r#	Defaults	Requires DCL
32-bit output Patch Constant Data Element (o#)	32 See note below	W	4	Yes	None	Yes

ⓘ Note

The hull shader fork and join phase outputs are a shared set of 4 4-vector registers. The outputs of each fork or join phase program cannot overlap with each other. System-interpreted values such as TessFactors come out of this space.

Related topics

[Shader Model 5](#)

Registers - ds_5_0

Article • 08/23/2019 • 2 minutes to read

The following input and output registers are implemented in the domain shader version 5_0.

Input Registers

Register Type	Count	R/W	Dimension	Indexable by r#	Defaults	Requires DCL
32-bit Temp (r#)	4096(r#+x#[n])	R/W	4	No	None	Yes
32-bit indexable Temp Array (x#[n])	4096(r#+x#[n])	R/W	4	Yes	None	Yes
32-bit Input Control Points (vcp[vertex][element])	32 See note 1 below.	R	4(component)*32(element)*32(vert)	Yes	None	Yes
32-bit Input Patch Constants (vpc[vertex])	32 See note 2 below.	R	4	Yes	None	Yes
32-bit input location in domain (vDomain.xy, vDomain.xyz))	1	R	3	No	N/A	Yes
32-bit UINT Input PrimitiveID (vPrim)	1	R	1	No	N/A	Yes
Element in an input resource (t#)	128	R	128	Yes	None	Yes
Sampler (s#)	16	R	1	Yes	None	Yes
iConstantBuffer reference (cb#[index])	15	R	4	Yes	None	Yes

Register Type	Count	R/W	Dimension	Indexable by r#	Defaults	Requires DCL
ilmmmediate ConstantBuffer reference (icb[index])	1	R	4	Yes(contents)	None	Yes

Note 1: The domain shader sees the hull shader outputs in 2 separate sets of registers. The vcp registers can see all of the hull shader's output Control Points. The vpc registers can see all of the hull shader's Patch Constant output data.

Note 2: Because code for hull shader Patch Constant Fork or Join Phases output TessFactors using names such as SV_TessFactor, the domain shader must match those declarations on the equivalent vpc input if it wishes to see those values.

Output Registers

Register Type	Count	R/W	Dimension	Indexable by r#	Defaults	Requires DCL
32-bit output Vertex Data Element (o#)	32	W	4	Yes	None	Yes

Related topics

[Shader Model 5](#)

Registers - cs_5_0

Article • 08/23/2019 • 2 minutes to read

The following input and output registers are implemented in the compute shader version 5_0.

Input Registers

Register Type	Count	R/W	Dimension	Indexable by r#	Defaults	Requires DCL
32-bit Temp (r#)	4096(r#+x#[n])	R/W	4	No	None	Yes
32-bit Indexable Temp Array (x#[n])	4096(r#+x#[n])	R/W	4	Yes	None	Yes
32-bit Thread Group Shared Memory (g#[n])	8192 (sum of all shared memory decls for thread group)	R/W	1 (can be declared various ways)	Yes	None	Yes
Element in an input resource (t#)	128	R	1	No	None	Yes
Sampler (s#)	16	R	1	No	None	Yes
ConstantBuffer reference (cb#[index])	15	R	4	Yes (contents)	None	Yes
Immediate ConstantBuffer reference (icb[index])	1	R	4	Yes(contents)	None	Yes
ThreadID (vThreadId.xyz)	1	R	3	No	N/A	Yes
ThreadGroupID (vThreadGroupID.xyz)	1	R	3	No	N/A	Yes
ThreadIdInGroup (vThreadIdInGroup.xyz)	1	R	3	No	N/A	Yes
ThreadIdInGroupFlattened (vThreadIdInGroupFlattened)	1	R	1	No	N/A	Yes

Output Registers

Register Type	Count	R/W	Dimension	Indexable by r#	Defaults	Requires DCL
NULL (discard result, useful for ops with multiple results)	N/A	W	N/A	N/A	N/A	No
Unordered Access View (u#)	8	R/W	1	No	No	Yes

Related topics

[Shader Model 5](#)

Shader Model 5.1

Article • 11/20/2019 • 2 minutes to read

This section contains the reference pages for HLSL Shader Model 5.1, introduced with D3D12.

Shader Model 5.1 is functionally very similar to [Shader Model 5](#), the main change is more flexibility in resource selection by allowing indexing of arrays of descriptors from within a shader.

Feature	Capability
Instruction Set	HLSL intrinsic functions
Vertex Shader Max	No restriction
Pixel Shader Max	No restriction
New Shader Profiles Added	cs_5_1, ds_5_1, gs_5_1, hs_5_1, ps_5_1, vs_5_1, rootsig_1_0

In this section

Topic	Description
Shader Model 5.1 Objects	The following objects have been added to shader model 5.1.
Shader Model 5.1 System Values	Shader Model 5.1 adds the following new system values.

Related topics

[Shader Models vs Shader Profiles](#)

[HLSL Shader Model 5.1 Features for Direct3D 12](#)

Shader Model 5.1 Objects

Article • 06/08/2021 • 2 minutes to read

The following objects have been added to shader model 5.1.

For the [Rasterizer Order Views](#) (available in D3D11.3 and D3D12), the following objects are new, and are only allowed in the pixel shader. Note that the methods they support are identical to the corresponding UAV objects.

Rasterizer object	UAV object
RasterizerOrderedBuffer	RWBuffer
RasterizerOrderedByteAddressBuffer	RWByteAddressBuffer
RasterizerOrderedStructuredBuffer	RWStructuredBuffer
RasterizerOrderedTexture1D	RWTexure1D
RasterizerOrderedTexture1DArray	RWTexure1DArray
RasterizerOrderedTexture2D	RWTexure2D
RasterizerOrderedTexture2DArray	RWTexure2DArray
RasterizerOrderedTexture3D	RWTexure3D

Related topics

[Shader Model 5.1](#)

[HSL Shader Model 5.1 Features for Direct3D 12](#)

Shader Model 5.1 System Values

Article • 08/23/2019 • 2 minutes to read

Shader Model 5.1 adds the following new system values.

- [SV_InnerCoverage](#)
- [SV_StencilRef](#)

Related topics

[Shader Model 5.1](#)

[HLSL Shader Model 5.1 Features for Direct3D 12](#)

SV_InnerCoverage

Article • 04/07/2022 • 2 minutes to read

SV_InnerCoverage represents underestimated conservative rasterization information (i.e. whether a pixel is guaranteed-to-be-fully covered).

Type

Type
uint

Remarks

This system generated value is required for tier 3 support, and is only available in that tier. This input is mutually exclusive with SV_Coverage - both cannot be used.

To access SV_InnerCoverage, it must be declared as a single component out of one of the pixel shader input registers. The interpolation mode on the declaration must be constant (interpolation does not apply).

Conservative rasterization is available in both D3D11.3 and D3D12. Refer to:

- [D3D11.3 Conservative Rasterization](#)
- [D3D12 Conservative Rasterization](#)

See also

[Shader Model 5](#)

[Shader Model 5.1 System Values](#)

SV_StencilRef

Article • 06/30/2021 • 2 minutes to read

SV_StencilRef represents the current pixel shader stencil reference value.

Type

Type
uint

Remarks

Specifying the shader reference value in the pixel shader is available in both D3D11.3 and D3D12. Refer to:

- [D3D11.3 Shader Specified Stencil Reference Value](#)
- [D3D12 Shader Specified Stencil Reference Value](#)

See also

[Shader Model 5](#)

[Shader Model 5.1 System Values](#)

Shader Model 6

Article • 11/04/2020 • 2 minutes to read

All non-quad related Wave Intrinsics are available in all shader stages. Quad wave intrinsics are available only in pixel and compute shaders.

In this section

Topic	Description
QuadReadAcrossDiagonal	Returns the specified local value which is read from the diagonally opposite lane in this quad.
QuadReadLaneAt	Returns the specified source value from the lane identified by the lane ID within the current quad.
QuadReadAcrossX	Returns the specified local value read from the other lane in this quad in the X direction.
QuadReadAcrossY	Returns the specified source value read from the other lane in this quad in the Y direction.
WaveActiveAllEqual	Returns true if the expression is the same for every active lane in the current wave (and thus uniform across it).
WaveActiveBitAnd	Returns the bitwise AND of all the values of the expression across all active lanes in the current wave and replicates it back to all active lanes.
WaveActiveBitOr	Returns the bitwise OR of all the values of the expression across all active lanes in the current wave and replicates it back to all active lanes.
WaveActiveBitXor	Returns the bitwise XOR of all the values of the expression across all active lanes in the current wave and replicates it back to all active lanes.
WaveActiveCountBits	Counts the number of boolean variables which evaluate to true across all active lanes in the current wave, and replicates the result to all lanes in the wave.
WaveActiveMax	Returns the maximum value of the expression across all active lanes in the current wave and replicates it back to all active lanes.
WaveActiveMin	Returns the minimum value of the expression across all active lanes in the current wave replicates it back to all active lanes.

Topic	Description
WaveActiveProduct	Multiplies the values of the expression together across all active lanes in the current wave and replicates it back to all active lanes.
WaveActiveSum	Sums up the value of the expression across all active lanes in the current wave and replicates it to all lanes in the current wave.
WaveActiveAllTrue	Returns true if the expression is true in all active lanes in the current wave.
WaveActiveAnyTrue	Returns true if the expression is true in any of the active lanes in the current wave.
WaveActiveBallot	Returns a 4-bit unsigned integer bitmask of the evaluation of the Boolean expression for all active lanes in the specified wave.
WaveGetLaneCount	Returns the number of lanes in a wave on this architecture.
WaveGetLaneIndex	Returns the index of the current lane within the current wave.
WavelsFirstLane	Returns true only for the active lane in the current wave with the smallest index.
WavePrefixCountBits	Returns the sum of all the specified boolean variables set to true across all active lanes with indices smaller than the current lane.
WavePrefixProduct	Returns the product of all of the values in the active lanes in this wave with indices less than this lane.
WavePrefixSum	Returns the sum of all of the values in the active lanes with smaller indices than this one.
WaveReadLaneFirst	Returns the value of the expression for the active lane of the current wave with the smallest index.
WaveReadLaneAt	Returns the value of the expression for the given lane index within the specified wave.

Related topics

[Overview of Shader Model 6](#)

[Shader Models vs Shader Profiles](#)

QuadReadAcrossDiagonal function

Article • 11/04/2020 • 2 minutes to read

Returns the specified local value which is read from the diagonally opposite lane in this quad.

Syntax

syntax

```
<type> QuadReadAcrossDiagonal(  
    <type> localValue  
>;
```

Parameters

localValue

The requested type.

Return value

The specified local value which is read from the diagonally opposite lane in this quad.

Remarks

For more information on quads, refer to [Overview of Shader Model 6](#).

This function is supported from shader model 6.0 only in pixel and compute shaders.

See also

[Shader Model 6](#)

QuadReadLaneAt function

Article • 11/04/2020 • 2 minutes to read

Returns the specified source value from the lane identified by the lane ID within the current quad.

Syntax

syntax

```
<type> QuadReadLaneAt(  
    <type> sourceValue,  
    uint    quadLaneID  
>;
```

Parameters

sourceValue

The requested type.

quadLaneID

The lane ID; this will be a value from 0 to 3.

Return value

The specified source value. The result of this function is uniform across the quad. If the source lane is inactive, the results are undefined.

Remarks

For more information on quads, refer to [Overview of Shader Model 6](#).

This function is supported from shader model 6.0 only in pixel and compute shaders.

See also

Shader Model 6

QuadReadAcrossX function

Article • 11/04/2020 • 2 minutes to read

Returns the specified local value read from the other lane in this quad in the X direction.

Syntax

syntax

```
<type> QuadReadAcrossX(  
    <type> localValue  
>;
```

Parameters

localValue

The requested type.

Return value

The specified local value. If the source lane is inactive, the results are undefined.

Remarks

For more information on quads, refer to [Overview of Shader Model 6](#).

This function is supported from shader model 6.0 only in pixel and compute shaders.

See also

[Shader Model 6](#)

QuadReadAcrossY function

Article • 11/04/2020 • 2 minutes to read

Returns the specified source value read from the other lane in this quad in the Y direction.

Syntax

syntax

```
<type> QuadReadAcrossY(  
    <type> localValue  
>;
```

Parameters

localValue

The requested type.

Return value

The specified source value. If the source lane is inactive, the results are undefined.

Remarks

For more information on quads, refer to [Overview of Shader Model 6](#).

This function is supported from shader model 6.0 only in pixel and compute shaders.

See also

[Shader Model 6](#)

WaveActiveAllEqual function

Article • 11/04/2020 • 2 minutes to read

Returns true if the expression is the same for every active lane in the current wave (and thus uniform across it).

Syntax

syntax

```
bool WaveActiveAllEqual(  
    <type> expr  
);
```

Parameters

expr

The expression to evaluate.

Return value

True if the expression is the same for every active lane in the current wave.

Remarks

This function is supported from shader model 6.0 in all shader stages.

See also

[Overview of Shader Model 6](#)

[Shader Model 6](#)

WaveActiveBitAnd function

Article • 11/04/2020 • 2 minutes to read

Returns the bitwise AND of all the values of the expression across all active lanes in the current wave and replicates it back to all active lanes.

Syntax

syntax

```
<int_type> WaveActiveBitAnd(  
    <int_type> expr  
>;
```

Parameters

expr

The expression to evaluate.

Return value

The bitwise AND value.

Remarks

This function is supported from shader model 6.0 in all shader stages.

See also

[Overview of Shader Model 6](#)

[Shader Model 6](#)

WaveActiveBitOr function

Article • 11/04/2020 • 2 minutes to read

Returns the bitwise OR of all the values of the expression across all active lanes in the current wave and replicates it back to all active lanes.

Syntax

syntax

```
<int_type> WaveActiveBitOr(  
    <int_type> expr  
>;
```

Parameters

expr

The expression to evaluate.

Return value

The bitwise OR value.

Remarks

This function is supported from shader model 6.0 in all shader stages.

See also

[Overview of Shader Model 6](#)

[Shader Model 6](#)

WaveActiveBitXor function

Article • 11/04/2020 • 2 minutes to read

Returns the bitwise XOR of all the values of the expression across all active lanes in the current wave and replicates it back to all active lanes.

Syntax

syntax

```
<int_type> WaveActiveBitXor(  
    <int_type> expr  
>;
```

Parameters

expr

The expression to evaluate.

Return value

The bitwise XOR value.

Remarks

This function is supported from shader model 6.0 in all shader stages.

See also

[Overview of Shader Model 6](#)

[Shader Model 6](#)

WaveActiveCountBits function

Article • 08/24/2021 • 2 minutes to read

Counts the number of boolean variables which evaluate to true across all active lanes in the current wave, and replicates the result to all lanes in the wave.

Syntax

syntax

```
uint WaveActiveCountBits(  
    bool bBit  
) ;
```

Parameters

bBit

The boolean variables to evaluate. Providing an explicit true Boolean value returns the number of active lanes.

Return value

The number of lanes for which the boolean variable evaluates to true, across all active lanes in the current wave.

Remarks

This function is supported from shader model 6.0 in all shader stages.

Examples

This can be implemented more efficiently than a full WaveActiveSum, as described in the following example:

syntax

```
result = WaveActiveCountBits( WaveActiveBallot( bBit ) );
```

See also

[Overview of Shader Model 6](#)

[Shader Model 6](#)

WaveActiveMax function

Article • 11/05/2020 • 2 minutes to read

Returns the maximum value of the expression across all active lanes in the current wave and replicates it back to all active lanes.

Syntax

syntax

```
<type> WaveActiveMax(  
    <type> expr  
>;
```

Parameters

expr

The expression to evaluate.

Return value

The maximum value.

Remarks

The order of operations is undefined.

This function is supported from shader model 6.0 in all shader stages.

Examples

syntax

```
float3 maxPos = WaveActiveMax( myPoint.position );  
BoundingBox.max = max( maxPos, BoundingBox.max );
```

See also

[Overview of Shader Model 6](#)

[Shader Model 6](#)

WaveActiveMin function

Article • 11/04/2020 • 2 minutes to read

Returns the minimum value of the expression across all active lanes in the current wave replicates it back to all active lanes.

Syntax

syntax

```
<type> WaveActiveMin(  
    <type> expr  
>;
```

Parameters

expr

The expression to evaluate.

Return value

The minimum value.

Remarks

The order of operations is undefined.

This function is supported from shader model 6.0 in all shader stages.

Examples

syntax

```
float3 minPos = WaveActiveMin( myPoint.position );  
BoundingBox.min = min( minPos, BoundingBox.min );
```

See also

[Overview of Shader Model 6](#)

[Shader Model 6](#)

WaveActiveProduct function

Article • 11/04/2020 • 2 minutes to read

Multiplies the values of the expression together across all active lanes in the current wave and replicates it back to all active lanes.

Syntax

syntax

```
<type> WaveActiveProduct(  
    <type> expr  
>;
```

Parameters

expr

The expression to evaluate.

Return value

The product value.

Remarks

The order of operations is undefined.

This function is supported from shader model 6.0 in all shader stages.

See also

[Overview of Shader Model 6](#)

[Shader Model 6](#)

WaveActiveSum function

Article • 11/04/2020 • 2 minutes to read

Sums up the value of the expression across all active lanes in the current wave and replicates it to all lanes in the current wave.

Syntax

syntax

```
<type> WaveActiveSum(  
    <type> expr  
>;
```

Parameters

expr

The expression to evaluate.

Return value

The sum value.

Remarks

The order of operations is undefined.

This function is supported from shader model 6.0 in all shader stages.

Examples

syntax

```
float3 total = WaveActiveSum( position ); // sum positions in wave  
float3 center = total/count;           // compute average of these positions
```

See also

[Overview of Shader Model 6](#)

[Shader Model 6](#)

WaveActiveAllTrue function

Article • 11/04/2020 • 2 minutes to read

Returns true if the expression is true in all active lanes in the current wave.

Syntax

syntax

```
bool WaveActiveAllTrue(  
    bool expr  
) ;
```

Parameters

expr

The boolean expression to evaluate.

Return value

True if the expression is true in all lanes.

Remarks

This function is supported from shader model 6.0 in all shader stages.

See also

[Overview of Shader Model 6](#)

[Shader Model 6](#)

WaveActiveAnyTrue function

Article • 11/04/2020 • 2 minutes to read

Returns true if the expression is true in any of the active lanes in the current wave.

Syntax

syntax

```
bool WaveActiveAnyTrue(  
    bool expr  
) ;
```

Parameters

expr

The boolean expression to evaluate.

Return value

True if the expression is true in any lane.

Remarks

This function is supported from shader model 6.0 in all shader stages.

See also

[Overview of Shader Model 6](#)

[Shader Model 6](#)

WaveActiveBallot function

Article • 11/04/2020 • 2 minutes to read

Returns a uint4 containing a bitmask of the evaluation of the Boolean expression for all active lanes in the current wave.

Syntax

syntax

```
uint4 WaveBallot(  
    bool expr  
);
```

Parameters

expr

The boolean expression to evaluate.

Return value

A uint4 containing a bitmask of the evaluation of the Boolean expression for all active lanes in the current wave. The least-significant bit corresponds to the lane with index zero. The bits corresponding to inactive lanes will be zero. The bits that are greater than or equal to [WaveGetLaneCount](#) will be zero.

Remarks

Different GPUs have different SIMD processor widths (lane counts). Most of these **WaveXXX** functions are able to operate at level of abstraction where SIMD machine width is concealed. To maximize portability of code across GPUs, use the intrinsics that don't rely on machine width. For example, use:

syntax

```
uint result = WaveActiveCountBits( bBit );
```

Instead of:

syntax

```
uint result = countbits( WaveActiveBallot( bBit ) );
```

This function is supported from shader model 6.0 in all shader stages.

Examples

syntax

```
// get a bitwise representation of the number of currently active lanes:  
uint4 waveBits = WaveActiveBallot( true ); // convert to bits
```

See also

[Overview of Shader Model 6](#)

[Shader Model 6](#)

WaveGetLaneCount function

Article • 11/04/2020 • 2 minutes to read

Returns the number of lanes in a wave on this architecture.

Syntax

syntax

```
uint WaveGetLaneCount(void);
```

Parameters

This function has no parameters.

Return value

The result will be between 4 and 128, and includes all waves: active, inactive, and/or helper lanes. The result returned from this function may vary significantly depending on the driver implementation.

Remarks

This function is supported from shader model 6.0 in all shader stages.

Examples

syntax

```
uint laneCount = WaveGetLaneCount();      // number of lanes in wave
```

See also

[Overview of Shader Model 6](#)

[Shader Model 6](#)

WaveGetLaneIndex function

Article • 11/04/2020 • 2 minutes to read

Returns the index of the current lane within the current wave.

Syntax

syntax

```
uint WaveGetLaneIndex(void);
```

Parameters

This function has no parameters.

Return value

The current lane index. The result will be between 0 and the result returned from [WaveGetLaneCount](#).

Remarks

This function is supported from shader model 6.0 in all shader stages.

See also

[Overview of Shader Model 6](#)

[Shader Model 6](#)

WaveIsFirstLane function

Article • 11/04/2020 • 2 minutes to read

Returns true only for the active lane in the current wave with the smallest index.

Syntax

syntax

```
bool WaveIsFirstLane(void);
```

Parameters

This function has no parameters.

Return value

True only for the active lane in the current wave with the smallest index.

Remarks

This function can be used to identify operations that are to be executed only once per wave.

This function is supported from shader model 6.0 in all shader stages.

Examples

syntax

```
if ( WaveIsFirstLane() )
{
    . . . // once per-wave code
}
```

See also

Overview of Shader Model 6

Shader Model 6

WavePrefixCountBits function

Article • 11/04/2020 • 2 minutes to read

Returns the sum of all the specified boolean variables set to true across all active lanes with indices smaller than the current lane.

Syntax

syntax

```
uint WavePrefixCountBits(  
    bool bBit  
) ;
```

Parameters

bBit

The specified boolean variables.

Return value

The sum of all the specified Boolean variables set to true across all active lanes with indices smaller than the current lane.

Remarks

This function is supported from shader model 6.0 in all shader stages.

Examples

The following code describes how to implement a compacted write to an ordered stream where the number of elements written per lane is either 1 or 0.

syntax

```
bool bDoesThisLaneHaveAnAppendItem = <expr>;  
// compute number of items to append for the whole wave
```

```
uint laneAppendOffset = WavePrefixCountBits( bDoesThisLaneHaveAnAppendItem );
);
uint appendCount = WaveActiveCountBits( bDoesThisLaneHaveAnAppendItem );
// update the output location for this whole wave
uint appendOffset;
if ( WaveIsFirstLane () )
{
    // this way, we only issue one atomic for the entire wave, which reduces
    contention
    // and keeps the output data for each lane in this wave together in the
    output buffer
    InterlockedAdd(bufferSize, appendCount, appendOffset);
}
appendOffset = WaveReadLaneFirst( appendOffset ); // broadcast value
appendOffset += laneAppendOffset; // and add in the offset for this lane
buffer[appendOffset] = myData; // write to the offset location for this lane
```

See also

[Overview of Shader Model 6](#)

[Shader Model 6](#)

WavePrefixProduct function

Article • 11/04/2020 • 2 minutes to read

Returns the product of all of the values in the active lanes in this wave with indices less than this lane.

Syntax

syntax

```
<type> WavePrefixProduct(  
    <type> value  
>;
```

Parameters

value

The value to multiply.

Return value

The product of all the values.

Remarks

The order of operations on this routine cannot be guaranteed. So, effectively, the [precise] flag is ignored within it.

A postfix product can be computed by multiplying the prefix product by the current lane's value.

Note that the active lane with the lowest index will always receive a 1 for its prefix product.

This function is supported from shader model 6.0 in all shader stages.

Examples

hlsl

```
uint numToMultiply = 2;  
uint prefixProduct = WavePrefixProduct( numToMultiply );
```

On a machine with a wave size of 8, and all lanes active except lanes 0 and 4, the following values would be returned from WavePrefixProduct.

lane index	status	prefixProduct
0	inactive	n/a
1	active	= 1
2	active	= 1*2
3	active	= 1*2*2
4	inactive	n/a
5	active	= 1*2*2*2
6	active	= 1*2*2*2*2
7	active	= 1*2*2*2*2*2

See also

[Overview of Shader Model 6](#)

[Shader Model 6](#)

WavePrefixSum function

Article • 01/30/2021 • 2 minutes to read

Returns the sum of all of the values in the active lanes with smaller indices than this one.

Syntax

syntax

```
<type> WavePrefixSum(  
    <type> value  
) ;
```

Parameters

value

The value to sum up.

Return value

The sum of the values.

Remarks

The order of operations on this routine cannot be guaranteed. So, effectively, the [precise] flag is ignored within it.

A postfix sum can be computed by adding the prefix sum to the current lane's value.

Note that the active lane with the lowest index will always receive a 0 for its prefix sum.

This function is supported from shader model 6.0 in all shader stages.

Examples

hlsl

```
uint numToSum = 2;  
uint prefixSum = WavePrefixSum( numToSum );
```

On a machine with a wave size of 8, and all lanes active except lanes 0 and 4, the following values would be returned from WavePrefixSum.

lane index	status	prefixSum
0	inactive	n/a
1	active	= 0
2	active	= 0+2
3	active	= 0+2+2
4	inactive	n/a
5	active	= 0+2+2+2
6	active	= 0+2+2+2+2
7	active	= 0+2+2+2+2+2

See also

[Overview of Shader Model 6](#)

[Shader Model 6](#)

WaveReadLaneFirst function

Article • 11/04/2020 • 2 minutes to read

Returns the value of the expression for the active lane of the current wave with the smallest index.

Syntax

syntax

```
<type> WaveReadLaneFirst(  
    <type> expr  
>;
```

Parameters

expr

The expression to evaluate.

Return value

The resulting value is uniform across the wave.

Remarks

This function is supported from shader model 6.0 in all shader stages.

See also

[Overview of Shader Model 6](#)

[Shader Model 6](#)

WaveReadLaneAt function

Article • 07/06/2021 • 2 minutes to read

Returns the value of the expression for the given lane index within the specified wave.

Syntax

```
syntax

<type> WaveReadLaneAt(
    <type> expr,
    uint laneIndex
);
```

Parameters

expr

The expression to evaluate.

laneIndex

The index of the lane for which the *expr* result will be returned.

Return value

The resulting value is the result of *expr*. It will be uniform if *laneIndex* is uniform.

Remarks

This function is effectively a broadcast of the value in the *laneIndex*'th lane.

This function is supported from shader model 6.0 in all shader stages.

See also

- [Overview of Shader Model 6](#)
- [Shader Model 6](#)

Intrinsic Functions

Article • 06/08/2022 • 7 minutes to read

The following table lists the intrinsic functions available in HLSL. Each function has a brief description, and a link to a reference page that has more detail about the input argument and return type.

Name	Description	Minimum shader model
abort	Terminates the current draw or dispatch call being executed.	4
abs	Absolute value (per component).	1 ¹
acos	Returns the arccosine of each component of x.	1 ¹
all	Test if all components of x are nonzero.	1 ¹
AllMemoryBarrier	Blocks execution of all threads in a group until all memory accesses have been completed.	5
AllMemoryBarrierWithGroupSync	Blocks execution of all threads in a group until all memory accesses have been completed and all threads in the group have reached this call.	5
any	Test if any component of x is nonzero.	1 ¹
asdouble	Reinterprets a cast value into a double.	5
asfloat	Convert the input type to a float.	4
asin	Returns the arcsine of each component of x.	1 ¹
asint	Convert the input type to an integer.	4
asuint	Reinterprets the bit pattern of a 64-bit type to a uint.	5
asuint	Convert the input type to an unsigned integer.	4
atan	Returns the arctangent of x.	1 ¹

Name	Description	Minimum shader model
<code>atan2</code>	Returns the arctangent of two values (x,y).	1 ¹
<code>ceil</code>	Returns the smallest integer which is greater than or equal to x.	1 ¹
<code>CheckAccessFullyMapped</code>	Determines whether all values from a Sample or Load operation accessed mapped tiles in a tiled resource .	5
<code>clamp</code>	Clamps x to the range [min, max].	1 ¹
<code>clip</code>	Discards the current pixel, if any component of x is less than zero.	1 ¹
<code>cos</code>	Returns the cosine of x.	1 ¹
<code>cosh</code>	Returns the hyperbolic cosine of x.	1 ¹
<code>countbits</code>	Counts the number of bits (per component) in the input integer.	5
<code>cross</code>	Returns the cross product of two 3D vectors.	1 ¹
<code>D3DCOLORtoUBYTE4</code>	Swizzles and scales components of the 4D vector x to compensate for the lack of UBYTE4 support in some hardware.	1 ¹
<code>ddx</code>	Returns the partial derivative of x with respect to the screen-space x-coordinate.	2 ¹
<code>ddx_coarse</code>	Computes a low precision partial derivative with respect to the screen-space x-coordinate.	5
<code>ddx_fine</code>	Computes a high precision partial derivative with respect to the screen-space x-coordinate.	5
<code>ddy</code>	Returns the partial derivative of x with respect to the screen-space y-coordinate.	2 ¹
<code>ddy_coarse</code>	Computes a low precision partial derivative with respect to the screen-space y-coordinate.	5

Name	Description	Minimum shader model
ddy_fine	Computes a high precision partial derivative with respect to the screen-space y-coordinate.	5
degrees	Converts x from radians to degrees.	1 ¹
determinant	Returns the determinant of the square matrix m.	1 ¹
DeviceMemoryBarrier	Blocks execution of all threads in a group until all device memory accesses have been completed.	5
DeviceMemoryBarrierWithGroupSync	Blocks execution of all threads in a group until all device memory accesses have been completed and all threads in the group have reached this call.	5
distance	Returns the distance between two points.	1 ¹
dot	Returns the dot product of two vectors.	1
dst	Calculates a distance vector.	5
errorf	Submits an error message to the information queue.	4
EvaluateAttributeCentroid	Evaluates at the pixel centroid.	5
EvaluateAttributeAtSample	Evaluates at the indexed sample location.	5
EvaluateAttributeSnapped	Evaluates at the pixel centroid with an offset.	5
exp	Returns the base-e exponent.	1 ¹
exp2	Base 2 exponent (per component).	1 ¹
f16tof32	Converts the float16 stored in the low-half of the uint to a float.	5
f32tof16	Converts an input into a float16 type.	5
faceforward	Returns -n * sign(dot(i, ng)).	1 ¹
firstbithigh	Gets the location of the first set bit starting from the highest order bit and working downward, per component.	5

Name	Description	Minimum shader model
firstbitlow	Returns the location of the first set bit starting from the lowest order bit and working upward, per component.	5
floor	Returns the greatest integer which is less than or equal to x.	1^1
fma	Returns the double-precision fused multiply-addition of $a * b + c$.	5
fmod	Returns the floating point remainder of x/y .	1^1
frac	Returns the fractional part of x.	1^1
frexp	Returns the mantissa and exponent of x.	2^1
fwidth	Returns $\text{abs}(\text{ddx}(x)) + \text{abs}(\text{ddy}(x))$	2^1
GetRenderTargetSampleCount	Returns the number of render-target samples.	4
GetRenderTargetSamplePosition	Returns a sample position (x,y) for a given sample index.	4
GroupMemoryBarrier	Blocks execution of all threads in a group until all group shared accesses have been completed.	5
GroupMemoryBarrierWithGroupSync	Blocks execution of all threads in a group until all group shared accesses have been completed and all threads in the group have reached this call.	5
InterlockedAdd	Performs a guaranteed atomic add of value to the dest resource variable.	5
InterlockedAnd	Performs a guaranteed atomic and.	5
InterlockedCompareExchange	Atomically compares the input to the comparison value and exchanges the result.	5
InterlockedCompareStore	Atomically compares the input to the comparison value.	5
InterlockedExchange	Assigns value to dest and returns the original value.	5
InterlockedMax	Performs a guaranteed atomic max.	5

Name	Description	Minimum shader model
InterlockedMin	Performs a guaranteed atomic min.	5
InterlockedOr	Performs a guaranteed atomic or.	5
InterlockedXor	Performs a guaranteed atomic xor.	5
isfinite	Returns true if x is finite, false otherwise.	1 ¹
isinf	Returns true if x is +INF or -INF, false otherwise.	1 ¹
isnan	Returns true if x is NAN or QNAN, false otherwise.	1 ¹
Idexp	Returns $x * 2^{\text{exp}}$	1 ¹
length	Returns the length of the vector v.	1 ¹
lerp	Returns $x + s(y - x)$.	1 ¹
lit	Returns a lighting vector (ambient, diffuse, specular, 1)	1 ¹
log	Returns the base-e logarithm of x.	1 ¹
log10	Returns the base-10 logarithm of x.	1 ¹
log2	Returns the base-2 logarithm of x.	1 ¹
mad	Performs an arithmetic multiply/add operation on three values.	5
max	Selects the greater of x and y.	1 ¹
min	Selects the lesser of x and y.	1 ¹
modf	Splits the value x into fractional and integer parts.	1 ¹
msad4	Compares a 4-byte reference value and an 8-byte source value and accumulates a vector of 4 sums.	5
mul	Performs matrix multiplication using x and y.	1
noise	Generates a random value using the Perlin-noise algorithm.	1 ¹

Name	Description	Minimum shader model
normalize	Returns a normalized vector.	1 ¹
pow	Returns x^y .	1 ¹
printf	Submits a custom shader message to the information queue.	4
Process2DQuadTessFactorsAvg	Generates the corrected tessellation factors for a quad patch.	5
Process2DQuadTessFactorsMax	Generates the corrected tessellation factors for a quad patch.	5
Process2DQuadTessFactorsMin	Generates the corrected tessellation factors for a quad patch.	5
ProcessIsolineTessFactors	Generates the rounded tessellation factors for an isoline.	5
ProcessQuadTessFactorsAvg	Generates the corrected tessellation factors for a quad patch.	5
ProcessQuadTessFactorsMax	Generates the corrected tessellation factors for a quad patch.	5
ProcessQuadTessFactorsMin	Generates the corrected tessellation factors for a quad patch.	5
ProcessTriTessFactorsAvg	Generates the corrected tessellation factors for a tri patch.	5
ProcessTriTessFactorsMax	Generates the corrected tessellation factors for a tri patch.	5
ProcessTriTessFactorsMin	Generates the corrected tessellation factors for a tri patch.	5
radians	Converts x from degrees to radians.	1
rcp	Calculates a fast, approximate, per-component reciprocal.	5
reflect	Returns a reflection vector.	1
refract	Returns the refraction vector.	1 ¹
reversebits	Reverses the order of the bits, per component.	5

Name	Description	Minimum shader model
round	Rounds x to the nearest integer	1 ¹
rsqrt	Returns $1 / \sqrt{x}$	1 ¹
saturate	Clamps x to the range [0, 1]	1
sign	Computes the sign of x.	1 ¹
sin	Returns the sine of x	1 ¹
sincos	Returns the sine and cosine of x.	1 ¹
sinh	Returns the hyperbolic sine of x	1 ¹
smoothstep	Returns a smooth Hermite interpolation between 0 and 1.	1 ¹
sqrt	Square root (per component)	1 ¹
step	Returns $(x \geq a) ? 1 : 0$	1 ¹
tan	Returns the tangent of x	1 ¹
tanh	Returns the hyperbolic tangent of x	1 ¹
tex1D(s, t)	1D texture lookup.	1
tex1D(s, t, ddx, ddy)	1D texture lookup.	2 ¹
tex1Dbias	1D texture lookup with bias.	2 ¹
tex1Dgrad	1D texture lookup with a gradient.	2 ¹
tex1Dlod	1D texture lookup with LOD.	3 ¹
tex1Dproj	1D texture lookup with projective divide.	2 ¹
tex2D(s, t)	2D texture lookup.	1 ¹
tex2D(s, t, ddx, ddy)	2D texture lookup.	2 ¹
tex2Dbias	2D texture lookup with bias.	2 ¹
tex2Dgrad	2D texture lookup with a gradient.	2 ¹
tex2Dlod	2D texture lookup with LOD.	3
tex2Dproj	2D texture lookup with projective divide.	2 ¹

Name	Description	Minimum shader model
<code>tex3D(s, t)</code>	3D texture lookup.	1^1
<code>tex3D(s, t, ddx, ddy)</code>	3D texture lookup.	2^1
<code>tex3Dbias</code>	3D texture lookup with bias.	2^1
<code>tex3Dgrad</code>	3D texture lookup with a gradient.	2^1
<code>tex3Dlod</code>	3D texture lookup with LOD.	3^1
<code>tex3Dproj</code>	3D texture lookup with projective divide.	2^1
<code>texCUBE(s, t)</code>	Cube texture lookup.	1^1
<code>texCUBE(s, t, ddx, ddy)</code>	Cube texture lookup.	2^1
<code>texCUBEbias</code>	Cube texture lookup with bias.	2^1
<code>texCUBEgrad</code>	Cube texture lookup with a gradient.	2^1
<code>texCUBElod</code>	Cube texture lookup with LOD.	3^1
<code>texCUBEproj</code>	Cube texture lookup with projective divide.	2^1
<code>transpose</code>	Returns the transpose of the matrix m.	1
<code>trunc</code>	Truncates floating-point value(s) to integer value(s)	1

¹ see reference page for restrictions.

Component and Template Types

The HLSL intrinsic function declarations use component types and template types for input parameter arguments and return values. The available types are listed in the following table.

These Template Types	Description	Support These Data Types
matrix	up to 16 components depending on the declaration	Basic HLSL Types

These Template Types	Description	Support These Data Types
object	sampler object	<i>sampler, sampler1D, sampler2D, sampler3D, samplerCUBE</i>
scalar	1 component	Basic HLSL Types
vector	1 component minimum, 4 components maximum (inclusive)	Basic HLSL Types

See also

[Reference for HLSL](#)

abort function

Article • 03/15/2021 • 2 minutes to read

Submits an error message to the information queue and terminates the current draw or dispatch call being executed.

Syntax

```
syntax  
  
void abort()  
);
```

Parameters

None

Return value

This function does not return a value.

Remarks

This operation does nothing on rasterizers that do not support it.

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 4 (DirectX HLSL) or later.	yes

Requirements

Requirement	Value

Requirement	Value
Header	Corecrt_terminate.h

See also

[Intrinsic Functions](#)

abs

Article • 08/19/2020 • 2 minutes to read

Returns the absolute value of the specified value.

```
ret abs(x)
```

Parameters

Item	Description
x	[in] The specified value.

Return Value

The absolute value of the x parameter.

Type Description

Name	Template Type	Component Type	Size
x	scalar, vector, or matrix	float, int	any
ret	same as input x	same as input x	same dimension(s) as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	vs_1_1 and ps_1_4

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

acos

Article • 08/19/2020 • 2 minutes to read

Returns the arccosine of the specified value.

```
ret acos(x)
```

Parameters

Item	Description
x	[in] The specified value. Each component should be a floating-point value within the range of -1 to 1.

Return Value

The arccosine of the x parameter.

Type Description

Name	Template Type	Component Type	Size
x	scalar , vector , or matrix	float	any
ret	same as input x	float	same dimension(s) as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	vs_1_1 only

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

all

Article • 08/19/2020 • 2 minutes to read

Determines if all components of the specified value are non-zero.

```
ret all(x)
```

Parameters

Item	Description
x	[in] The specified value.

Return Value

True if all components of the x parameter are non-zero; otherwise, false.

Remarks

This function is similar to the [any](#) HLSL intrinsic function. The [any](#) function determines if any components of the specified value are non-zero, while the [all](#) function determines if all components of the specified value are non-zero.

Type Description

Name	Template Type	Component Type	Size
x	scalar , vector , or matrix	float , int , bool	any
ret	scalar	bool	1

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes

Shader Model	Supported
Shader Model 1 (DirectX HLSL)	vs_1_1 and ps_1_4

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

AllMemoryBarrier function

Article • 10/24/2019 • 2 minutes to read

Blocks execution of all threads in a group until all memory accesses have been completed.

Syntax

syntax

```
void AllMemoryBarrier(void);
```

Parameters

This function has no parameters.

Return value

This function does not return a value.

Remarks

A memory barrier guarantees that outstanding memory operations have completed. Threads are synchronized at GroupSync barriers. This may stall a thread or threads if memory operations are in progress.

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

AllMemoryBarrierWithGroupSync function

Article • 10/24/2019 • 2 minutes to read

Blocks execution of all threads in a group until all memory accesses have been completed and all threads in the group have reached this call.

Syntax

Syntax

```
void AllMemoryBarrierWithGroupSync(void);
```

Parameters

This function has no parameters.

Return value

This function does not return a value.

Remarks

A memory barrier guarantees that outstanding memory operations have completed. Threads are synchronized at GroupSync barriers. This may stall a thread or threads if memory operations are in progress.

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

any

Article • 08/19/2020 • 2 minutes to read

Determines if any components of the specified value are non-zero.

```
ret any(x)
```

Parameters

Item	Description
x	[in] The specified value.

Return Value

True if any components of the x parameter are non-zero; otherwise, `false`.

Remarks

This function is similar to the [all](#) HLSL intrinsic function. The `any` function determines if any components of the specified value are non-zero, while the `all` function determines if all components of the specified value are non-zero.

Type Description

Name	Template Type	Component Type	Size
x	scalar , vector , or matrix	float , int , bool	any
ret	scalar	bool	1

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes

Shader Model	Supported
Shader Model 1 (DirectX HLSL)	vs_1_1 and ps_1_4

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

asddouble function

Article • 10/24/2019 • 2 minutes to read

Reinterprets a cast value (two 32-bit values) into a double.

Syntax

syntax

```
double asddouble(  
    in uint lowbits,  
    in uint highbits  
);
```

Parameters

lowbits [in]

Type: **uint**

The low 32-bit pattern of the input value.

highbits [in]

Type: **uint**

The high 32-bit pattern of the input value.

Return value

Type: **double**

The input (two 32-bit values) recast as a double.

Remarks

The following overloaded version is also available:

syntax

```
double2 asddouble(uint2 lowbits, uint2 highbits);
```

If the input value is two 32-bit components, the return type will contain one double. If the input value is four 32-bit components, the return type will contain two doubles. If the input value is a 64-bit type, the returned value will have the same number of components as the input value.

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

asfloat

Article • 08/25/2021 • 2 minutes to read

Interprets the bit pattern of x as a floating-point number.

```
ret asfloat(x)
```

Parameters

Item	Description
x	[in] The input value.

Return Value

The input interpreted as a floating-point number.

Type Description

Name	Template Type	Component Type	Size
x	scalar, vector, or matrix	float, int, uint	any
ret	same as input x	float	same dimension(s) as input x

Function Overloads

```
`float<x> asfloat(float<x> value);` `float<x> asfloat(int<x> value);` `float<x> asfloat(uint<x> value);`
```

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 4 and higher shader models	yes
Shader Model 3 (DirectX HLSL)	no

Shader Model	Supported
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Remarks

Older compilers incorrectly allowed `asfloat(bool)`, but note that bool inputs are not supported.

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

asin

Article • 08/19/2020 • 2 minutes to read

Returns the arcsine of the specified value.

```
ret asin(x)
```

Parameters

Item	Description
x	[in] The specified value.

Return Value

The arcsine of the *x* parameter.

Remarks

Each component of the *x* parameter should be within the range of $-\pi/2$ to $\pi/2$.

Type Description

Name	Template Type	Component Type	Size
x	scalar , vector , or matrix	float	any
ret	same as input x	float	same dimension(s) as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	vs_1_1

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

asint function

Article • 10/24/2019 • 2 minutes to read

Interprets the bit pattern of an input value as an integer. For more information about the **asint** intrinsic function, see [asint \(DirectX HLSL\)](#).

Syntax

syntax

```
int asint(  
    in  value  
) ;
```

Parameters

value [in]

The input value.

Return value

Type: **int**

The input interpreted as an integer.

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

asint

Article • 08/19/2020 • 2 minutes to read

Interprets the bit pattern of x as an integer.

```
ret asint(x)
```

Parameters

Item	Description
x	[in] The input value.

Return Value

The input interpreted as an integer.

Type Description

Name	Template Type	Component Type	Size
x	scalar, vector, or matrix	float, uint	any
ret	same as input x	int	same dimension(s) as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 4 and higher shader models	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

See also

Intrinsic Functions (DirectX HLSL)

asuint function

Article • 10/24/2019 • 2 minutes to read

Reinterprets the bit pattern of a 64-bit value as two unsigned 32-bit integers.

Syntax

syntax

```
void asuint(  
    in double value,  
    out uint lowbits,  
    out uint highbits  
) ;
```

Parameters

value [in]

Type: **double**

The input value.

lowbits [out]

Type: **uint**

The low 32-bit pattern of *value*.

highbits [out]

Type: **uint**

The high 32-bit pattern of *value*.

Return value

This function does not return a value.

Remarks

This function is an alternate version of the [asuint](#) intrinsic that has been available in earlier shader models, and was introduced for Shader Model 5. The original function (recognized in the HLSL compiler by its different signature) remains available to Shader Model 5.

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Intrinsic Functions](#)

[asuint \(DirectX HLSL\)](#)

[Shader Model 5](#)

asuint

Article • 08/19/2020 • 2 minutes to read

Interprets the bit pattern of *x* as an unsigned integer.

```
ret asuint(x)
```

Parameters

Item	Description
<i>x</i>	[in] The input value.

Return Value

The input interpreted as an unsigned integer.

Type Description

Name	Template Type	Component Type	Size
<i>x</i>	scalar , vector , or matrix	float , int	any
<i>ret</i>	same as input <i>x</i>	uint	same dimension(s) as input <i>x</i>

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 4 and higher shader models	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

See also

Intrinsic Functions (DirectX HLSL)

atan

Article • 08/19/2020 • 2 minutes to read

Returns the arctangent of the specified value.

```
ret atan(x)
```

Parameters

Item	Description
x	[in] The specified value.

Return Value

The arctangent of the *x* parameter. This value is within the range of $-\pi/2$ to $\pi/2$.

Type Description

Name	Template Type	Component Type	Size
x	scalar, vector, or matrix	float	any
ret	same as input x	float	same dimension(s) as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	vs_1_1

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

atan2

Article • 03/15/2021 • 2 minutes to read

Returns the arctangent of two values (x,y).

```
ret atan2(y, x)
```

Parameters

Item	Description
y	[in] The y value.
x	[in] The x value.

Return Value

The arctangent of (y,x).

Remarks

The signs of the *x* and *y* parameters are used to determine the quadrant of the return values within the range of $-\pi$ to π . The **atan2** HLSL intrinsic function is well-defined for every point other than the origin, even if *y* equals 0 and *x* does not equal 0.

Type Description

Name	Template Type	Component Type	Size
y	same as input x	float	same dimension(s) as input x
x	scalar, vector, or matrix	float	any
ret	same as input x	float	same dimension(s) as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	vs_1_1

Requirements

Requirement	Value
Header	Corecrt_math.h

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

ceil

Article • 03/15/2021 • 2 minutes to read

Returns the smallest integer value that is greater than or equal to the specified value.

```
ret ceil(x)
```

Parameters

Item	Description
x	[in] The specified value.

Return Value

The smallest integer value (returned as a floating-point type) that is greater than or equal to the x parameter.

Type Description

Name	Template Type	Component Type	Size
x	scalar, vector, or matrix	float	any
ret	same as input x	float	same dimension(s) as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	vs_1_1

Requirements

Requirement	Value

Requirement	Value
Header	Corecrt_math.h

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

CheckAccessFullyMapped function

Article • 08/19/2020 • 2 minutes to read

Determines whether all values from a **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#).

Syntax

syntax

```
bool CheckAccessFullyMapped(  
    in uint_only status  
>);
```

Parameters

status [in]

Type: **uint_only**

The status value that is returned from a **Sample**, **Gather**, or **Load** operation. Because you can't access this status value directly, you need to pass it to **CheckAccessFullyMapped**.

Return value

Type: **bool**

Returns a **Boolean** value that indicates whether all values from a **Sample**, **Gather**, or **Load** operation accessed mapped tiles in a [tiled resource](#). Returns **TRUE** if all values from the operation accessed mapped tiles; otherwise, returns **FALSE** if any values were taken from an unmapped tile.

Remarks

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
--------------	-----------

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

See also

[Intrinsic Functions](#)

clamp

Article • 08/19/2020 • 2 minutes to read

Clamps the specified value to the specified minimum and maximum range.

```
ret clamp(x, min, max)
```

Parameters

Item	Description
<i>x</i>	[in] A value to clamp.
<i>min</i>	[in] The specified minimum range.
<i>max</i>	[in] The specified maximum range.

Return Value

The clamped value for the *x* parameter.

Remarks

For values of -INF or INF, clamp will behave as expected. However for values of NaN, the results are undefined.

Type Description

Name	Template Type	Component Type	Size
<i>x</i>	scalar , vector , or matrix	float , int	any
<i>min</i>	same as input <i>x</i>	float , int	same dimension(s) as input <i>x</i>
<i>max</i>	same as input <i>x</i>	float , int	same dimension(s) as input <i>x</i>
<i>ret</i>	same as input <i>x</i>	float , int	same dimension(s) as input <i>x</i>

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	vs_1_1 and ps_1_4

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

clip

Article • 08/19/2020 • 2 minutes to read

Discards the current pixel if the specified value is less than zero.

```
clip(x)
```

Parameters

Item	Description
x	[in] The specified value.

Return Value

None.

Remarks

Use the `clip` HLSL intrinsic function to simulate clipping planes if each component of the `x` parameter represents the distance from a plane.

Also, use the `clip` function to test for alpha behavior, as shown in the following example:

```
clip( Input.Color.A < 0.1f ? -1:1 );
```

Type Description

Name	Template Type	Component Type	Size
x	<code>scalar</code> , <code>vector</code> , or <code>matrix</code>	<code>float</code>	any

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 4	yes (pixel shader only)
Shader Model 3 (DirectX HLSL)	yes (pixel shader only)
Shader Model 2 (DirectX HLSL)	yes (pixel shader only)
Shader Model 1 (DirectX HLSL)	yes (pixel shader only)

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

COS

Article • 08/19/2020 • 2 minutes to read

Returns the cosine of the specified value.

```
ret cos(x)
```

Parameters

Item	Description
x	[in] The specified value, in radians.

Return Value

The cosine of the x parameter.

Type Description

Name	Template Type	Component Type	Size
x	scalar, vector, or matrix	float	any
ret	same as input x	float	same dimension(s) as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	vs_1_1

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

cosh

Article • 08/19/2020 • 2 minutes to read

Returns the hyperbolic cosine of the specified value.

```
ret cosh(x)
```

Parameters

Item	Description
x	[in] The specified value, in radians.

Return Value

The hyperbolic cosine of the x parameter.

Type Description

Name	Template Type	Component Type	Size
x	scalar, vector, or matrix	float	any
ret	same as input x	float	same dimension(s) as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	vs_1_1

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

countbits function

Article • 04/22/2021 • 2 minutes to read

Counts the number of bits (per component) set in the input integer.

Syntax

syntax

```
uint countbits(  
    in uint value  
) ;
```

Parameters

value [in]

Type: **uint**

The input value.

Return value

Type: **uint**

The number of bits.

Remarks

The following overloaded versions are also available:

syntax

```
uint count_bits(uint value);  
uint2 count_bits(uint2 value);  
uint3 count_bits(uint3 value);  
uint4 count_bits(uint4 value);
```

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

CROSS

Article • 08/19/2020 • 2 minutes to read

Returns the cross product of two floating-point, 3D vectors.

```
ret cross(x, y)
```

Parameters

Item	Description
x	[in] The first floating-point, 3D vector.
y	[in] The second floating-point, 3D vector.

Return Value

The cross product of the x parameter and the y parameter.

Type Description

Name	Template Type	Component Type	Size
x	vector	float	3
y	vector	float	3
ret	vector	float	3

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	vs_1_1 and ps_1_4

See also

Intrinsic Functions (DirectX HLSL)

D3DCOLORtoUBYTE4

Article • 08/19/2020 • 2 minutes to read

Converts a floating-point, 4D vector set by a D3DCOLOR to a UBYTE4.

```
ret D3DCOLORtoUBYTE4(x)
```

This function swizzles and scales components of the *x* parameter. Use this function to compensate for the lack of UBYTE4 support in some hardware.

Parameters

Item	Description
<i>x</i>	[in] The floating-point vector4 to convert.

Return Value

The UBYTE4 representation of the *x* parameter.

Type Description

Name	Template Type	Component Type	Size
<i>x</i>	vector	float	4
<i>ret</i>	vector	integer	4

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	vs_1_1

See also

Intrinsic Functions (DirectX HLSL)

ddx

Article • 08/19/2020 • 2 minutes to read

Returns the partial derivative of the specified value with respect to the screen-space x-coordinate.

```
ret ddx(x)
```

This function computes the partial derivative with respect to the screen-space x-coordinate. To compute the partial derivative with respect to the screen-space y-coordinate, use the [ddy](#) function.

This function is only supported in pixel shaders.

Parameters

Item	Description
x	[in] The specified value.

Return Value

The partial derivative of the x parameter.

Type Description

Name	Template Type	Component Type	Size
x	scalar , vector , or matrix	float	any
ret	same as input x	float	same dimension(s) as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

Shader Model	Supported
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	yes
Shader Model 2 (DirectX HLSL)	yes in ps_2_x; unsupported in ps_2_0.
Shader Model 1 (DirectX HLSL)	no

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

ddx_coarse function

Article • 10/24/2019 • 2 minutes to read

Computes a low precision partial derivative with respect to the screen-space x-coordinate.

Syntax

syntax

```
float ddx_coarse(  
    in float value  
) ;
```

Parameters

value [in]

Type: **float**

The input value.

Return value

Type: **float**

The low precision partial derivative of *value*.

Remarks

The following overloaded versions are also available:

syntax

```
float2 ddx_coarse(float2 value);  
float3 ddx_coarse(float3 value);  
float4 ddx_coarse(float4 value);
```

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

ddx_fine function

Article • 03/09/2021 • 2 minutes to read

Computes a high precision partial derivative with respect to the screen-space x-coordinate.

Syntax

syntax

```
float ddx_fine(  
    in float value  
) ;
```

Parameters

value [in]

Type: **float**

The input value.

Return value

Type: **float**

The high precision partial derivative of *value*.

Remarks

The following overloaded versions are also available:

syntax

```
float2 ddx_fine(float2 value);  
float3 ddx_fine(float3 value);  
float4 ddx_fine(float4 value);
```

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

ddy

Article • 08/19/2020 • 2 minutes to read

Returns the partial derivative of the specified value with respect to the screen-space y-coordinate.

```
ret ddy(x)
```

This function computes the partial derivative with respect to the screen-space y-coordinate. To compute the partial derivative with respect to the screen-space x-coordinate, use the [ddx](#) function.

This function is only supported in pixel shaders.

Parameters

Item	Description
x	[in] The specified value.

Return Value

The partial derivative of the *x* parameter.

Type Description

Name	Template Type	Component Type	Size
x	scalar , vector , or matrix	float	any
ret	same as input x	float	same dimension(s) as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

Shader Model	Supported
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	yes
Shader Model 2 (DirectX HLSL)	yes in ps_2_x; unsupported in ps_2_0.
Shader Model 1 (DirectX HLSL)	no

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

ddy_coarse function

Article • 10/24/2019 • 2 minutes to read

Computes a low precision partial derivative with respect to the screen-space y-coordinate.

Syntax

syntax

```
float ddy_coarse(  
    in float value  
) ;
```

Parameters

value [in]

Type: **float**

The input value.

Return value

Type: **float**

The low precision partial derivative of *value*.

Remarks

The following overloaded versions are also available:

syntax

```
float2 ddy_coarse(float2 value);  
float3 ddy_coarse(float3 value);  
float4 ddy_coarse(float4 value);
```

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

ddy_fine function

Article • 03/09/2021 • 2 minutes to read

Computes a high precision partial derivative with respect to the screen-space x-coordinate.

Syntax

syntax

```
float ddy_fine(  
    in float value  
) ;
```

Parameters

value [in]

Type: **float**

The input value.

Return value

Type: **float**

The high precision partial derivative of *value*.

Remarks

The following overloaded versions are also available:

syntax

```
float2 ddy_fine(float2 value);  
float3 ddy_fine(float3 value);  
float4 ddy_fine(float4 value);
```

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

degrees

Article • 08/19/2020 • 2 minutes to read

Converts the specified value from radians to degrees.

```
ret degrees(x)
```

Parameters

Item	Description
x	[in] The specified value.

Return Value

The result of converting the *x* parameter from radians to degrees.

Type Description

Name	Template Type	Component Type	Size
x	scalar, vector, or matrix	float	any
ret	same as input x	float	same dimension(s) as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	vs_1_1

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

determinant

Article • 08/19/2020 • 2 minutes to read

Returns the determinant of the specified floating-point, square matrix.

```
ret determinant(m)
```

Parameters

Item	Description
<i>m</i>	[in] The specified value.

Return Value

The floating-point, scalar value that represents the determinant of the *m* parameter.

Type Description

Name	Template Type	Component Type	Size
<i>m</i>	matrix	float	any (number of rows = number of columns)
ret	scalar	float	1

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	vs_1_1 and ps_1_4

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

DeviceMemoryBarrier function

Article • 10/24/2019 • 2 minutes to read

Blocks execution of all threads in a group until all device memory accesses have been completed.

Syntax

syntax

```
void DeviceMemoryBarrier(void);
```

Parameters

This function has no parameters.

Return value

This function does not return a value.

Remarks

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

DeviceMemoryBarrierWithGroupSync function

Article • 10/24/2019 • 2 minutes to read

Blocks execution of all threads in a group until all device memory accesses have been completed and all threads in the group have reached this call.

Syntax

Syntax

```
void DeviceMemoryBarrierWithGroupSync(void);
```

Parameters

This function has no parameters.

Return value

This function does not return a value.

Remarks

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
					x

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

distance

Article • 08/19/2020 • 2 minutes to read

Returns a distance scalar between two vectors.

```
ret distance(x, y)
```

Parameters

Item	Description
x	[in] The first floating-point vector to compare.
y	[in] The second floating-point vector to compare.

Return Value

A floating-point, scalar value that represents the distance between the *x* parameter and the *y* parameter.

Type Description

Name	Template Type	Component Type	Size
x	vector	float	any
y	vector	float	same dimension(s) as input x
ret	scalar	float	1

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	vs_1_1

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

dot

Article • 08/19/2020 • 2 minutes to read

Returns the dot product of two vectors.

```
ret dot(x, y)
```

Parameters

Item	Description
x	[in] The first vector.
y	[in] The second vector.

Return Value

The dot product of the x parameter and the y parameter.

Type Description

Name	Template Type	Component Type	Size
x	vector	float, int	any
y	vector	float, int	same dimensions(s) as input x
ret	scalar	float, int	1

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	yes

See also

Intrinsic Functions (DirectX HLSL)

dst function

Article • 03/09/2021 • 2 minutes to read

Calculates a distance vector.

Syntax

syntax

```
fVector dst(  
    in fVector src0,  
    in fVector src1  
) ;
```

Parameters

src0 [in]

Type: [fVector](#)

The first vector.

src1 [in]

Type: [fVector](#)

The second vector.

Return value

Type: [fVector](#)

The computed distance vector.

Remarks

This intrinsic function provides the same functionality as the Vertex Shader instruction [dst](#).

See also

Intrinsic Functions

errorf function

Article • 10/24/2019 • 2 minutes to read

Submits an error message to the information queue.

Syntax

syntax

```
void errorf(  
    string format,  
    argument ...  
) ;
```

Parameters

format

The format string.

argument ...

Optional arguments.

Return value

This function does not return a value.

Remarks

This operation does nothing on devices that do not support it.

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 4 (DirectX HLSL) or later.	yes

See also

[Intrinsic Functions](#)

EvaluateAttributeCentroid function

Article • 06/08/2022 • 2 minutes to read

Evaluates at the pixel centroid.

Syntax

syntax

```
numeric EvaluateAttributeCentroid(  
    in attrib numeric value  
) ;
```

Parameters

value [in]

Type: **attrib numeric**

The input value.

Remarks

Interpolation mode can be **linear** or **linear_no_perspective** on the variable. Use of **centroid** or **sample** is ignored. Attributes with constant interpolation are also allowed, in which case the sample index is ignored.

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
--------	------	--------	----------	-------	---------

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

EvaluateAttributeAtSample function

Article • 10/24/2019 • 2 minutes to read

Evaluates at the indexed sample location.

Syntax

```
syntax

numeric EvaluateAttributeAtSample(
    in attrib numeric value,
    in uint sampleindex
);
```

Parameters

value [in]

Type: **attrib numeric**

The input value.

sampleindex [in]

Type: **uint**

The sample location.

Remarks

Interpolation mode can be **linear** or **linear_no_perspective** on the variable. Use of **centroid** or **sample** is ignored. Attributes with constant interpolation are also allowed, in which case the sample index is ignored.

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

EvaluateAttributeSnapped function

Article • 10/24/2019 • 2 minutes to read

Evaluates at the pixel centroid with an offset.

Syntax

```
syntax

numeric EvaluateAttributeSnapped(
    in attrib numeric value,
    in
        int2 offset
);
```

Parameters

value [in]

Type: **attrib numeric**

The input value.

offset [in]

Type: **int2**

A 2D offset from the pixel center using a 16x16 grid.

Remarks

The range for the *offset* parameter must be defined by the following byte code.

Only the least significant 4 bits of the first two components (U, V) of the pixel offset are used. The conversion from the 4-bit fixed point to float is as follows (MSB...LSB), where the MSB is both a part of the fraction and determines the sign:

- 1000 = -0.5f (-8 / 16)
- 1001 = -0.4375f (-7 / 16)
- 1010 = -0.375f (-6 / 16)
- 1011 = -0.3125f (-5 / 16)
- 1100 = -0.25f (-4 / 16)

- 1101 = -0.1875f (-3 / 16)
- 1110 = -0.125f (-2 / 16)
- 1111 = -0.0625f (-1 / 16)
- 0000 = 0.0f (0 / 16)
- 0001 = 0.0625f (1 / 16)
- 0010 = 0.125f (2 / 16)
- 0011 = 0.1875f (3 / 16)
- 0100 = 0.25f (4 / 16)
- 0101 = 0.3125f (5 / 16)
- 0110 = 0.375f (6 / 16)
- 0111 = 0.4375f (7 / 16)

Note

The left and top edges of a pixel are included in the offset; however, the bottom and right edges are not included. All other bits in the 32-bit integer U and V offset values are ignored.

An implementation can take the offset provided by the shader and obtain a full 32-bit fixed point value (28.4), which spans the valid range, by performing the following calculation:

```
iU = (iU<<28)>>28 // keep lowest 4 bits and sign extend, which yields  
[-8..7]
```

If an implementation must map the offset to a floating-point offset, it performs the following calculation:

```
fU = ((float)iU)/16
```

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
--------------	-----------

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				x	

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

exp

Article • 08/19/2020 • 2 minutes to read

Returns the base-e exponential, or e^x , of the specified value.

```
ret exp(x)
```

Parameters

Item	Description
x	[in] The specified value.

Return Value

The base-e exponential of the x parameter.

Type Description

Name	Template Type	Component Type	Size
x	scalar, vector, or matrix	float	any
ret	same as input x	float	same dimension(s) as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	vs_1_1

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

exp2

Article • 03/15/2021 • 2 minutes to read

Returns the base 2 exponential, or 2^x , of the specified value.

```
ret exp2(x)
```

Parameters

Item	Description
x	[in] The specified floating-point value.

Return Value

The base 2 exponential of the *x* parameter.

Type Description

Name	Template Type	Component Type	Size
x	scalar , vector , or matrix	float	any
ret	same as input x	float	same dimension(s) as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	vs_1_1

Requirements

Requirement	Value

Requirement	Value
Header	Corecrt_math.h

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

f16tof32 function

Article • 10/24/2019 • 2 minutes to read

Converts the float16 stored in the low-half of the uint to a float.

Syntax

syntax

```
float f16tof32(  
    in uint value  
) ;
```

Parameters

value [in]

Type: **uint**

The input value.

Return value

Type: **float**

The converted value.

Remarks

The following overloaded versions are also available:

syntax

```
float2 f16tof32(uint2 value);  
float3 f16tof32(uint3 value);  
float4 f16tof32(uint4 value);
```

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 4 and higher shader models	yes

Although this function is available in [Shader Model 4](#) and higher shader models, because it is emulated in 4.0 and 4.1, it is less performant on these shader models than it is on [Shader Model 5](#).

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

f32tof16 function

Article • 10/24/2019 • 2 minutes to read

Converts an input into a float16 type.

Syntax

syntax

```
uint f32tof16(  
    in float value  
) ;
```

Parameters

value [in]

Type: **float**

The input value.

Return value

Type: **uint**

The converted value, stored in the low-half of the uint.

Remarks

The following overloaded versions are also available:

syntax

```
uint2 f32tof16(float2 value);  
uint3 f32tof16(float3 value);  
uint4 f32tof16(float4 value);
```

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 4 and higher shader models	yes

Although this function is available in [Shader Model 4](#) and higher shader models, because it is emulated in 4.0 and 4.1, it is less performant on these shader models than it is on [Shader Model 5](#).

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

faceforward

Article • 08/19/2020 • 2 minutes to read

Flips the surface-normal (if needed) to face in a direction opposite to i ; returns the result in n .

```
ret faceforward(n, i, ng)
```

This function uses the following formula: $-n \operatorname{sign}(\operatorname{dot}(i, ng))$.

Parameters

Item	Description
<i>n</i>	[in] The resulting floating-point surface-normal vector.
<i>i</i>	[in] A floating-point, incident vector that points from the view position to the shading position.
<i>ng</i>	[in] A floating-point surface-normal vector.

Return Value

A floating-point, surface normal vector that is facing the view direction.

Type Description

Name	Template Type	Component Type	Size
<i>n</i>	vector	float	any
<i>i</i>	vector	float	same dimension(s) as input <i>n</i>
<i>ng</i>	vector	float	same dimensions as input <i>n</i>
<i>ret</i>	vector	float	same dimensions as input <i>n</i>

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	vs_1_1 and ps_1_4

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

firstbithigh function

Article • 08/19/2020 • 2 minutes to read

Gets the location of the first set bit starting from the highest order bit and working downward, per component.

Syntax

syntax

```
int firstbithigh(  
    in int value  
>;
```

Parameters

value [in]

Type: [int](#)

The input value.

Return value

Type: [int](#)

The location of the first set bit.

Remarks

For a signed integer, the first significant bit is zero for a negative number.

The following overloaded versions are also available:

syntax

```
int2 firstbithigh(int2 value);  
int3 firstbithigh(int3 value);  
int4 firstbithigh(int4 value);  
uint firstbithigh(uint value);  
uint2 firstbithigh(uint2 value);
```

```
uint3 firstbithigh(uint3 value);
uint4 firstbithigh(uint4 value);
```

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

firstbitlow function

Article • 08/19/2020 • 2 minutes to read

Returns the location of the first set bit starting from the lowest order bit and working upward, per component.

Syntax

```
syntax  
  
int firstbitlow(  
    in int value  
);
```

Parameters

value [in]

Type: [int](#)

The input value.

Return value

Type: [int](#)

The location of the first set bit.

Remarks

The following overloaded versions are also available:

```
syntax  
  
uint2 firstbitlow(uint2 value);  
uint3 firstbitlow(uint3 value);  
uint4 firstbitlow(uint4 value);
```

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

floor

Article • 03/15/2021 • 2 minutes to read

Returns the largest integer that is less than or equal to the specified value.

```
ret floor(x)
```

Parameters

Item	Description
x	[in] The specified value.

Return Value

The largest integer value (returned as a floating-point type) that is less than or equal to the *x* parameter.

Type Description

Name	Template Type	Component Type	Size
x	scalar, vector, or matrix	float	any
ret	same as input x	float	same dimension(s) as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	vs_1_1

Requirements

Requirement	Value

Requirement	Value
Header	Corecrt_math.h

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

fma

Article • 12/11/2020 • 2 minutes to read

Returns the double-precision fused multiply-addition of $a * b + c$.

```
ret fma(double a, b, c);
```

Parameters

a

[in] The first value in the fused multiply-addition.

b

[in] The second value in the fused multiply-addition.

c

[in] The third value in the fused multiply-addition.

Return Value

The double-precision fused multiply-addition of parameters $a * b + c$. The returned value must be accurate to 0.5 units of least precision (ULP).

Remarks

The **fma** intrinsic must support NaNs, INFs, and Denorms.

To use the **fma** intrinsic in your shader code, call the [ID3D11Device::CheckFeatureSupport](#) method with [D3D11_FEATURE_D3D11_OPTIONS](#) to verify that the Direct3D device supports the [ExtendedDoublesShaderInstructions](#) feature option. The **fma** intrinsic requires a WDDM 1.2 display driver, and all WDDM 1.2 display drivers must support **fma**. If your app creates a rendering device with [feature level](#) 11.0 or 11.1 and the compilation target is shader model 5 or later, the HLSL source code can use the **fma** intrinsic.

Type Description

Name	Template Type	Component Type	Size
<i>a</i>	scalar, vector, or matrix	double	any
<i>b</i>	same as input <i>a</i>	double	same dimensions as input <i>a</i>
<i>c</i>	same as input <i>a</i>	double	same dimensions as input <i>a</i>
<i>ret</i>	same as input <i>a</i>	double	same dimensions as input <i>a</i>

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader model 5 or later	yes

Requirements

Requirement	Value
Minimum supported client	Windows 8 [desktop apps UWP apps]
Minimum supported server	Windows Server 2012 [desktop apps UWP apps]
Header	Corecrt_math.h

See also

[Intrinsic Functions](#)

fmod

Article • 03/15/2021 • 2 minutes to read

Returns the floating-point remainder of x/y .

```
ret fmod(x, y)
```

Parameters

Item	Description
x	[in] The floating-point dividend.
y	[in] The floating-point divisor.

Return Value

The floating-point remainder of the x parameter divided by the y parameter.

Remarks

The floating-point remainder is calculated such that $x = i * y + f$, where i is an integer, f has the same sign as x , and the absolute value of f is less than the absolute value of y .

Type Description

Name	Template Type	Component Type	Size
x	scalar , vector, or matrix	float	any
y	same as input x	float	same dimension(s) as input x
ret	same as input x	float	same dimension(s) as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	vs_1_1

Requirements

Requirement	Value
Header	Corecrt_math.h

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

frac

Article • 08/19/2020 • 2 minutes to read

Returns the fractional (or decimal) part of x ; which is greater than or equal to 0 and less than 1.

Also see [trunc](#).

```
ret frac(x)
```

Parameters

Item	Description
x	[in] The specified value.

Return Value

The fractional part of the x parameter.

Type Description

Name	Template Type	Component Type	Size
x	scalar , vector , or matrix	float	any
ret	same as input x	float	same dimension(s) as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	vs_1_1

See also

Intrinsic Functions (DirectX HLSL)

frexp

Article • 03/15/2021 • 2 minutes to read

Returns the mantissa and exponent of the specified floating-point value.

```
ret frexp(x, exp)
```

The return value is the mantissa, and the value returned by *exp* parameter is the exponent.

Parameters

Item	Description
<i>x</i>	[in] The specified floating-point value. If the <i>x</i> parameter is 0, this function returns 0 for both the mantissa and the exponent.
<i>exp</i>	[out] The returned exponent of the <i>x</i> parameter.

Return Value

The mantissa of the *x* parameter.

Type Description

Name	Template Type	Component Type	Size
<i>x</i>	scalar , vector , or matrix	float	any
<i>exp</i>	same as input <i>x</i>	float	same dimension(s) as input <i>x</i>
<i>ret</i>	same as input <i>x</i>	float	same dimension(s) as input <i>x</i>

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 3 (DirectX HLSL) and higher shader models	yes

Shader Model	Supported
Shader Model 2 (DirectX HLSL)	yes (ps_2_x only)
Shader Model 1 (DirectX HLSL)	no

Remarks

Requirements

Requirement	Value
Header	Corecrt_math.h

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

fwidth

Article • 08/19/2020 • 2 minutes to read

Returns the absolute value of the partial derivatives of the specified value.

```
ret fwidth(x)
```

This function computes the following: `abs(ddx(x)) + abs(ddy(x))`.

This function is only supported in pixel shaders.

Parameters

Item	Description
<code>x</code>	[in] The specified value.

Return Value

The absolute value of the partial derivatives of the `x` parameter.

Type Description

Name	Template Type	Component Type	Size
<code>x</code>	<code>scalar</code> , <code>vector</code> , or <code>matrix</code>	<code>float</code>	any
<code>ret</code>	same as input <code>x</code>	<code>float</code>	same dimension(s) as input <code>x</code>

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 3 (DirectX HLSL) and higher shader models	yes
Shader Model 2 (DirectX HLSL)	yes (ps_2_x only)
Shader Model 1 (DirectX HLSL)	no

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

GetRenderTargetSampleCount

Article • 10/24/2019 • 2 minutes to read

Gets the number of samples for a render target.

```
UINT GetRenderTargetSampleCount()
```

Parameters

Item	Description
None	

Return Value

The number of samples.

Remarks

Use this function and [GetRenderTargetSamplePosition](#) to find out the number and position of the sampling locations for a render target.

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 4 and higher shader models	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

GetRenderTargetSamplePosition

Article • 06/30/2021 • 2 minutes to read

Gets the sampling position (x,y) for a given sample index.

```
float<2> GetRenderTargetSamplePosition( in int<1> Index );
```

Parameters

Item	Description
<i>Index</i>	[in] A zero-based sample index.

Return Value

The (x,y) position of the given sample.

Remarks

Use this function and [GetRenderTargetSampleCount](#) to find out the number and position of the sampling locations for a render target.

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 4 and higher shader models	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

GroupMemoryBarrier function

Article • 10/24/2019 • 2 minutes to read

Blocks execution of all threads in a group until all group shared accesses have been completed.

Syntax

syntax

```
void GroupMemoryBarrier(void);
```

Parameters

This function has no parameters.

Return value

This function does not return a value.

Remarks

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
					x

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

GroupMemoryBarrierWithGroupSync function

Article • 10/24/2019 • 2 minutes to read

Blocks execution of all threads in a group until all group shared accesses have been completed and all threads in the group have reached this call.

Syntax

Syntax

```
void GroupMemoryBarrierWithGroupSync(void);
```

Parameters

This function has no parameters.

Return value

This function does not return a value.

Remarks

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
					x

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

InterlockedAdd function (HLSL reference)

Article • 06/29/2022 • 2 minutes to read

Performs a guaranteed atomic add of value to the dest resource variable.

Syntax

syntax

```
void InterlockedAdd(
    in R dest,
    in T value,
    out T original_value
);
```

Parameters

dest [in]

Type: R

The destination address.

value [in]

Type: T

The input value.

original_value [out]

Type: T

Optional. The original input value.

Return value

This function does not return a value.

Remarks

This operation can only be performed on int or uint typed resources and shared memory variables. There are two possible uses for this function. The first is when R is a shared memory variable type. In this case, the function performs an atomic add of value to the shared memory register referenced by dest. The second scenario is when R is a resource variable type. In this scenario, the function performs an atomic add of value to the resource location referenced by dest. The overloaded function has an additional output variable which will be set to the original value of dest. This overloaded operation is only available when R is readable and writable.

Interlocked operations do not imply any memory fence/barrier.

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

InterlockedAnd function (HLSL reference)

Article • 06/29/2022 • 2 minutes to read

Performs a guaranteed atomic and.

Syntax

syntax

```
void InterlockedAnd(
    in R dest,
    in T value,
    out T original_value
);
```

Parameters

dest [in]

Type: R

The destination address.

value [in]

Type: T

The input value.

original_value [out]

Type: T

Optional. The original input value.

Return value

This function does not return a value.

Remarks

This operation can only be performed on int or uint typed resources and shared memory variables. There are two possible uses for this function. The first is when R is a shared memory variable type. In this case, the function performs an atomic and of value to the shared memory register referenced by dest. The second scenario is when R is a resource variable type. In this scenario, the function performs an atomic and of value to the resource location referenced by dest. The overloaded function has an additional output variable which will be set to the original value of dest. This overloaded operation is only available when R is readable and writable.

Interlocked operations do not imply any memory fence/barrier.

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

InterlockedCompareExchange function (HLSL reference)

Article • 06/29/2022 • 2 minutes to read

Atomically compares the destination with the comparison value. If they are identical, the destination is overwritten with the input value. The original value is set to the destination's original value.

Syntax

syntax

```
void InterlockedCompareExchange(
    in R dest,
    in T compare_value,
    in T value,
    out T original_value
);
```

Parameters

dest [in]

Type: R

The destination address.

compare_value [in]

Type: T

The comparison value.

value [in]

Type: T

The input value.

original_value [out]

Type: T

The original value.

Return value

This function does not return a value.

Remarks

Atomically compares the value referenced by *dest* with *compare_value*, stores *value* in the location referenced by *dest* if the values match, returns the original value of *dest* in *original_value*. This operation can only be performed on **int** or **uint** typed resources and shared memory variables. There are two possible uses for this function. The first is when R is a shared memory variable type. In this case, the function performs the operation on the shared memory register referenced by *dest*. The second scenario is when R is a resource variable type. In this scenario, the function performs the operation on the resource location referenced by *dest*. This operation is only available when R is readable and writable.

Interlocked operations do not imply any memory fence/barrier.

ⓘ Note

If you call **InterlockedCompareExchange** in a **for** or **while** compute shader loop, to properly compile, you must use the **[allow_uav_condition]** attribute on that loop.

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
--------	------	--------	----------	-------	---------

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

InterlockedCompareStore function (HLSL reference)

Article • 06/29/2022 • 2 minutes to read

Atomically compares the destination to the comparison value. If they are identical, the destination is overwritten with the input value.

Syntax

Syntax

```
void InterlockedCompareStore(
    in R dest,
    in T compare_value,
    in T value
);
```

Parameters

dest [in]

Type: R

The destination address.

compare_value [in]

Type: T

The comparison value.

value [in]

Type: T

The input value.

Return value

This function does not return a value.

Remarks

Atomically compares the value referenced by *dest* with *compare_value* and stores *value* in the location referenced by *dest* if the values match. This operation can only be performed on **int** or **uint** typed resources and shared memory variables. There are two possible uses for this function. The first is when R is a shared memory variable type. In this case, the function performs the operation on the shared memory register referenced by *dest*. The second scenario is when R is a resource variable type. In this scenario, the function performs the operation on the resource location referenced by *dest*.

Interlocked operations do not imply any memory fence/barrier.

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

InterlockedExchange function (HLSL reference)

Article • 06/29/2022 • 2 minutes to read

Assigns value to dest and returns the original value.

Syntax

syntax

```
void InterlockedExchange(
    in R dest,
    in T value,
    out T original_value
);
```

Parameters

dest [in]

Type: R

The destination address.

value [in]

Type: T

The input value.

original_value [out]

Type: T

The original value.

Return value

This function does not return a value.

Remarks

This operation can only be performed on scalar-typed resources and shared memory variables. There are two possible uses for this function. The first is when R is a shared memory variable type. In this case, the function performs the operation on the shared memory register referenced by dest. The second scenario is when R is a resource variable type. In this scenario, the function performs the operation on the resource location referenced by dest. This operation is only available when R is readable and writable.

Interlocked operations do not imply any memory fence/barrier.

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

InterlockedMax function (HSL reference)

Article • 06/29/2022 • 2 minutes to read

Performs a guaranteed atomic max.

Syntax

syntax

```
void InterlockedMax(
    in R dest,
    in T value,
    out T original_value
);
```

Parameters

dest [in]

Type: R

The destination address.

value [in]

Type: T

The input value.

original_value [out]

Type: T

Optional. The original input value.

Return value

This function does not return a value.

Remarks

This operation can only be performed on int and uint typed resources and shared memory variables. There are two possible uses for this function. The first is when R is a shared memory variable type. In this case, the function performs an atomic max of value to the shared memory register referenced by dest. The second scenario is when R is a resource variable type. In this scenario, the function performs an atomic max of value to the resource location referenced by dest. The overloaded function has an additional output variable which will be set to the original value of dest. This overloaded operation is only available when R is readable and writable.

Interlocked operations do not imply any memory fence/barrier.

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

InterlockedMin function (HLSL reference)

Article • 06/29/2022 • 2 minutes to read

Performs a guaranteed atomic min.

Syntax

syntax

```
void InterlockedMin(  
    in R dest,  
    in T value,  
    out T original_value  
);
```

Parameters

dest [in]

Type: R

The destination address.

value [in]

Type: T

The input value.

original_value [out]

Type: T

Optional. The original input value.

Return value

This function does not return a value.

Remarks

This operation can only be performed on int and uint typed resources and shared memory variables. There are two possible uses for this function. The first is when R is a shared memory variable type. In this case, the function performs an atomic min of value to the shared memory register referenced by dest. The second scenario is when R is a resource variable type. In this scenario, the function performs an atomic min of value to the resource location referenced by dest. The overloaded function has an additional output variable which will be set to the original value of dest. This overloaded operation is only available when R is readable and writable.

Interlocked operations do not imply any memory fence/barrier.

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

InterlockedOr function (HLSL reference)

Article • 06/29/2022 • 2 minutes to read

Performs a guaranteed atomic or.

Syntax

syntax

```
void InterlockedOr(  
    in R dest,  
    in T value,  
    out T original_value  
) ;
```

Parameters

dest [in]

Type: **R**

The destination address.

value [in]

Type: **T**

The input value.

original_value [out]

Type: **T**

Optional. The original input value.

Return value

This function does not return a value.

Remarks

This operation can only be performed on int or uint typed resources and shared memory variables. There are two possible uses for this function. The first is when R is a shared memory variable type. In this case, the function performs an atomic or of value to the shared memory register referenced by dest. The second scenario is when R is a resource variable type. In this scenario, the function performs an atomic or of value to the resource location referenced by dest. The overloaded function has an additional output variable which will be set to the original value of dest. This overloaded operation is only available when R is readable and writable.

Interlocked operations do not imply any memory fence/barrier.

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

InterlockedXor function (HLSL reference)

Article • 06/29/2022 • 2 minutes to read

Performs a guaranteed atomic xor.

Syntax

syntax

```
void InterlockedXor(  
    in R dest,  
    in T value,  
    out T original_value  
>;
```

Parameters

dest [in]

Type: R

The destination address.

value [in]

Type: T

The input value.

original_value [out]

Type: T

Optional. The original input value.

Return value

This function does not return a value.

Remarks

This operation can only be performed on int or uint typed resources and shared memory variables. There are two possible uses for this function. The first is when R is a shared memory variable type. In this case, the function performs an atomic XOR of value to the shared memory register referenced by dest. The second scenario is when R is a resource variable type. In this scenario, the function performs an atomic XORof value to the resource location referenced by dest. The overloaded function has an additional output variable which will be set to the original value of dest. This overloaded operation is only available when R is readable and writable.

Interlocked operations do not imply any memory fence/barrier.

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

isfinite

Article • 03/15/2021 • 2 minutes to read

Determines if the specified floating-point value is finite.

```
ret isfinite(x)
```

Parameters

Item	Description
x	[in] The specified value.

Return Value

Returns a value of the same size as the input, with a value set to **True** if the *x* parameter is finite; otherwise **False**.

Type Description

Name	Template Type	Component Type	Size
x	scalar, vector, or matrix	float	any
ret	scalar, vector, or matrix	bool	as input

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	yes (vs_1_1 only)

Requirements

Requirement	Value

Requirement	Value
Header	Corecrt_math.h

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

isinf

Article • 03/15/2021 • 2 minutes to read

Determines if the specified value is infinite.

```
ret isinf(x)
```

Parameters

Item	Description
x	[in] The specified value.

Return Value

Returns a value of the same size as the input, with a value set to **True** if the *x* parameter is +INF or -INF. Otherwise, **False**.

Type Description

Name	Template Type	Component Type	Size
x	scalar, vector, or matrix	float	any
ret	scalar, vector, or matrix	bool	as input

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	yes (vs_1_1 only)

Requirements

Requirement	Value

Requirement	Value
Header	Corecrt_math.h

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

isnan

Article • 03/15/2021 • 2 minutes to read

Determines if the specified value is NAN or QNAN.

```
ret isnan(x)
```

Parameters

Item	Description
x	[in] The specified value.

Return Value

Returns a value of the same size as the input, with a value set to **True** if the *x* parameter is NAN or QNAN. Otherwise, **False**.

Type Description

Name	Template Type	Component Type	Size
x	scalar, vector, or matrix	float	any
ret	scalar, vector, or matrix	bool	as input

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	yes (vs_1_1 only)

Requirements

Requirement	Value

Requirement	Value
Header	Corecrt_math.h

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

Idexp

Article • 03/15/2021 • 2 minutes to read

Returns the result of multiplying the specified value by two, raised to the power of the specified exponent.

```
ret Idexp(x, exp)
```

This function uses the following formula: $x * 2^{\text{exp}}$

Parameters

Item	Description
<i>x</i>	[in] The specified value.
<i>exp</i>	[in] The specified exponent.

Return Value

The result of multiplying the *x* parameter by two, raised to the power of the *exp* parameter.

Type Description

Name	Template Type	Component Type	Size
<i>x</i>	scalar, vector, or matrix	float	any
<i>exp</i>	same as input <i>x</i>	float	same dimension(s) as input <i>x</i>
<i>ret</i>	same as input <i>x</i>	float	same dimension(s) as input <i>x</i>

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes

Shader Model	Supported
Shader Model 1 (DirectX HLSL)	yes (vs_1_1 only)

Requirements

Requirement	Value
Header	Corecrt_math.h

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

length

Article • 08/19/2020 • 2 minutes to read

Returns the length of the specified floating-point vector.

```
ret length(x)
```

Parameters

Item	Description
x	The specified floating-point vector.

Return Value

A floating-point scalar that represents the length of the *x* parameter.

Type Description

Name	Template Type	Component Type	Size
x	vector	float	any
ret	scalar	float	1

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	yes (vs_1_1 only)

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

lerp

Article • 08/19/2020 • 2 minutes to read

Performs a linear interpolation.

```
ret lerp(x, y, s)
```

Parameters

Item	Description
x	[in] The first-floating point value.
y	[in] The second-floating point value.
s	[in] A value that linearly interpolates between the x parameter and the y parameter.

Return Value

The result of the linear interpolation.

Type Description

Name	Template Type	Component Type	Size
x	scalar, vector, or matrix	float	any
y	same as input x	float	same dimension(s) as input x
s	same as input x	float	same dimension(s) as input x
ret	same as input x	float	same dimension(s) as input x

Remarks

Linear interpolation is based on the following formula: $x*(1-s) + y*s$ which can equivalently be written as $x + s(y-x)$.

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	yes (vs_1_1 and ps_1_1)

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

lit

Article • 08/19/2020 • 2 minutes to read

Returns a lighting coefficient vector.

```
ret lit(n_dot_l, n_dot_h, m)
```

This function returns a lighting coefficient vector (ambient, diffuse, specular, 1) where:

- ambient = 1
- diffuse = $n \cdot l < 0 ? 0 : n \cdot l$
- specular = $n \cdot l < 0 || n \cdot h < 0 ? 0 : (n \cdot h)^m$

Where the vector n is the normal vector, l is the direction to light and h is the half vector.

Parameters

Item	Description
n_dot_l	[in] The dot product of the normalized surface normal and the light vector.
n_dot_h	[in] The dot product of the half-angle vector and the surface normal.
m	[in] A specular exponent.

Return Value

The lighting coefficient vector.

Type Description

Name	Template Type	Component Type	Size
n_dot_l	scalar	float	1
n_dot_h	scalar	float	1
m	scalar	float	1
ret	vector	float	4

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	yes (vs_1_1 only)

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

log

Article • 08/19/2020 • 2 minutes to read

Returns the base-e logarithm of the specified value.

```
ret log(x)
```

Parameters

Item	Description
x	[in] The specified value.

Return Value

The base-e logarithm of the *x* parameter. If the *x* parameter is negative, this function returns indefinite. If the *x* parameter is 0, this function returns -INF.

Type Description

Name	Template Type	Component Type	Size
x	scalar, vector, or matrix	float	any
ret	same as input x	float	same dimension(s) as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	yes (vs_1_1 only)

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

log10

Article • 08/19/2020 • 2 minutes to read

Returns the base-10 logarithm of the specified value.

```
ret log10(x)
```

Parameters

Item	Description
x	[in] The specified value.

Return Value

The base-10 logarithm of the x parameter. If the x parameter is negative, this function returns indefinite. If the x is 0, this function returns -INF.

Type Description

Name	Template Type	Component Type	Size
x	scalar, vector, or matrix	float	any
ret	same as input x	float	same dimension(s) as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	yes (vs_1_1 only)

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

log2

Article • 03/15/2021 • 2 minutes to read

Returns the base-2 logarithm of the specified value.

```
ret log2(x)
```

Parameters

Item	Description
x	[in] The specified value.

Return Value

The base-2 logarithm of the *x* parameter. If the *x* parameter is negative, this function returns indefinite. If the *x* parameter is 0, this function returns +INF.

Type Description

Name	Template Type	Component Type	Size
x	scalar, vector, or matrix	float	any
ret	same as input x	float	same dimension(s) as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	yes (vs_1_1 only)

Requirements

Requirement	Value

Requirement	Value
Header	Corecrt_math.h

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

mad function

Article • 10/24/2019 • 2 minutes to read

Performs an arithmetic multiply/add operation on three values.

Syntax

syntax

```
numeric mad(  
    in numeric mvalue,  
    in numeric avalue,  
    in numeric bvalue  
) ;
```

Parameters

mvalue [in]

Type: **numeric**

The multiplication value.

avalue [in]

Type: **numeric**

The first addition value.

bvalue [in]

Type: **numeric**

The second addition value.

Return value

Type: **numeric**

The result of *mvalue* * *avalue* + *bvalue*.

Remarks

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

Shader authors can use the **mad** intrinsic to explicitly target the **mad** hardware instruction in the compiled shader output, which is particularly useful with shaders that mark results with the **precise** keyword. The **mad** instruction can be implemented in hardware as either "fused," which offers higher precision than implementing a **mul** instruction followed by an **add** instruction, or as a **mul + add**.

If shader authors use the **mad** intrinsic to calculate a result that the shader marked as precise, they indicate to the hardware to use any valid implementation of the **mad** instruction (fused or not) as long as the implementation is consistent for all uses of that **mad** intrinsic in any shader on that hardware. Shaders can then take advantage of potential performance improvements by using a native **mad** instruction (versus **mul + add**) on some hardware. The result of performing a native **mad** hardware instruction might or might not be different than performing a **mul** followed by an **add**. However, whatever the result is, the result must be consistent for the same operation to occur in multiple shaders or different parts of a shader.

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

max

Article • 08/19/2020 • 2 minutes to read

Selects the greater of x and y.

```
ret max(x, y)
```

Parameters

Item	Description
x	[in] The x input value.
y	[in] The y input value.

Return Value

The x or y parameter, whichever is the largest value.

Remarks

Denormals are handled as follows:

src0 src1->	-inf	F	+inf	NAN
-inf	-inf	src1	+inf	-inf
F	src0	src0 or src1	+inf	src0
+inf	+inf	+inf	+inf	+inf
NaN	-inf	src1	+inf	NaN

F means finite-real number.

Type Description

Name	In/Out	Template Type	Component Type	Size
x	in	scalar, vector, or matrix	float, int	any

Name	In/Out	Template Type	Component Type	Size
y	in	same as input x	float , int	same dimension(s) as input x
ret	return type	same as input x	float , int	same dimension(s) as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	yes (vs_1_1 and ps_1_4)

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

[DirectX Functional Specification](#) ↗

min

Article • 08/19/2020 • 2 minutes to read

Selects the lesser of x and y.

```
ret min(x, y)
```

Parameters

Item	Description
x	[in] The x input value.
y	[in] The y input value.

Return Value

The x or y parameter, whichever is the smallest value.

Remarks

Denormals are handled as follows:

src0 src1->	-inf	F	+inf	NAN
-inf	-inf	-inf	-inf	-inf
F	-inf	src0 or src1	src0	src0
+inf	-inf	src1	+inf	+inf
NaN	-inf	src1	+inf	NaN

F means finite-real number.

Type Description

Name	In/Out	Template Type	Component Type	Size
x	in	scalar, vector, or matrix	float, int	any

Name	In/Out	Template Type	Component Type	Size
y	in	same as input x	float , int	same dimension(s) as input x
ret	return type	same as input x	float , int	same dimension(s) as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	yes (vs_1_1 and ps_1_4)

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

[DirectX Functional Specification](#) ↗

modf

Article • 03/15/2021 • 2 minutes to read

Splits the value x into fractional and integer parts, each of which has the same sign as x .

```
ret modf(x, out ip)
```

Parameters

Item	Description
x	[in] The x input value.
ip	[out] The integer portion of x .

Return Value

The signed-fractional portion of x .

Type Description

Name	In/Out	Template Type	Component Type	Size
x	in	scalar, vector, or matrix	float, int	any
ip	out	same as input x	float, int	same dimension(s) as input x
ret	out	same as input x	float, int	same dimension(s) as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	yes (vs_1_1 only)

Requirements

Requirement	Value
Header	Corecrt_math.h

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

msad4

Article • 12/11/2020 • 2 minutes to read

Compares a 4-byte reference value and an 8-byte source value and accumulates a vector of 4 sums. Each sum corresponds to the masked sum of absolute differences of a different byte alignment between the reference value and the source value.

```
uint4 result = msad4(uint reference, uint2 source, uint4 accum);
```

Parameters

reference

[in] The reference array of 4 bytes in one **uint** value.

source

[in] The source array of 8 bytes in two **uint2** values.

accum

[in] A vector of 4 values. **msad4** adds this vector to the masked sum of absolute differences of the different byte alignments between the reference value and the source value.

Return Value

A vector of 4 sums. Each sum corresponds to the masked sum of absolute differences of different byte alignments between the reference value and the source value. **msad4** doesn't include a difference in the sum if that difference is masked (that is, the reference byte is 0).

Remarks

To use the **msad4** intrinsic in your shader code, call the

[ID3D11Device::CheckFeatureSupport](#) method with [D3D11_FEATURE_D3D11_OPTIONS](#) to verify that the Direct3D device supports the [SAD4ShaderInstructions](#) feature option.

The **msad4** intrinsic requires a WDDM 1.2 display driver, and all WDDM 1.2 display drivers must support **msad4**. If your app creates a rendering device with [feature level](#)

11.0 or 11.1 and the compilation target is shader model 5 or later, the HLSL source code can use the **msad4** intrinsic.

Return values are only accurate up to 65535. If you call the **msad4** intrinsic with inputs that might result in return values greater than 65535, **msad4** produces undefined results.

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader model 5 or later	yes

Examples

Here is an example result calculation for **msad4**:

```
reference = 0xA100B2C3;
source.x = 0xD7B0C372
source.y = 0x4F57C2A3
accum = {1,2,3,4}
result.x alignment source: 0xD7B0C372
result.x = accum.x + |0xD7    0xA1| + 0 (masked) + |0xC3    0xB2| + |0x72
0xC3| = 1 + 54 + 0 + 17 + 81 = 153
result.y alignment source: 0xA3D7B0C3
result.y = accum.y + |0xA3    0xA1| + 0 (masked) + |0xB0    0xB2| + |0xC3
0xC3| = 2 + 2 + 0 + 2 + 0 = 6
result.z alignment source: 0xC2A3D7B0
result.z = accum.z + |0xC2    0xA1| + 0 (masked) + |0xD7    0xB2| + |0xB0
0xC3| = 3 + 33 + 0 + 37 + 19 = 92
result.w alignment source: 0x57C2A3D7
result.w = accum.w + |0x57    0xA1| + 0 (masked) + |0xA3    0xB2| + |0xD7
0xC3| = 4 + 74 + 0 + 15 + 20 = 113
result = {153,6,92,113}
```

Here is an example of how you can use **msad4** to search for a reference pattern within a buffer:

```
uint4 accum = {0,0,0,0};
for(uint i=0;i<REF_SIZE;i++)
    accum = msad4(
```

```
buf_ref[i],  
    uint2(buf_src[DTid.x+i], buf_src[DTid.x+i+1]),  
    accum);  
buf_accum[DTid.x] = accum;
```

Requirements

Requirement	Value
Minimum supported client	Windows 8 [desktop apps UWP apps]
Minimum supported server	Windows Server 2012 [desktop apps UWP apps]

See also

[Intrinsic Functions](#)

mul

Article • 10/24/2019 • 2 minutes to read

Multiplies x and y using matrix math. The inner dimension x-columns and y-rows must be equal.

```
ret mul(x, y)
```

Parameters

Item	Description
x	[in] The x input value. If x is a vector, it treated as a row vector.
y	[in] The y input value. If y is a vector, it treated as a column vector.

Return Value

The result of x times y. The result has the dimension x-rows x y-columns.

Type Description

There are 9 overloaded versions of this function; the overloaded versions handle the different cases for the types and sizes of the input arguments.

Version	Name	Purpose	Template Type	Component Type	Size
1					
	x	in	scalar	float, int	1
	y	in	scalar	same as input x	1
	ret	out	scalar	same as input x	1
2					
	x	in	scalar	float, int	1
	y	in	vector	float, int	any

Version	Name	Purpose	Template Type	Component Type	Size
	ret	out	vector	float, int	same dimension(s) as input y
3	x	in	scalar	float, int	1
	y	in	matrix	float, int	any
	ret	out	matrix	same as input y	same dimension(s) as input y
4	x	in	vector	float, int	any
	y	in	scalar	float, int	1
	ret	out	vector	float, int	same dimension(s) as input x
5	x	in	vector	float, int	any
	y	in	vector	float, int	same dimension(s) as input x
	ret	out	scalar	float, int	1
6	x	in	vector	float, int	any
	y	in	matrix	float, int	rows = same dimension(s) as input x, columns = any
	ret	out	vector	float, int	same dimension(s) as input y columns
7	x	in	matrix	float, int	any
	y	in	scalar	float, int	1
	ret	out	matrix	float, int	same dimension(s) as input x
8	x	in	matrix	float, int	any
	y	in	vector	float, int	number of columns in input x

Version	Name	Purpose	Template Type	Component Type	Size
	ret	out	vector	float, int	number of rows in input x
9					
	x	in	matrix	float, int	any
	y	in	matrix	float, int	rows = number of columns in input x
	ret	out	matrix	float, int	rows = number of rows in input x, columns = number of columns in input y

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 1 (DirectX HLSL) and higher shader models	yes

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

noise

Article • 11/04/2020 • 2 minutes to read

Generates a random value using the Perlin-noise algorithm.

```
ret noise(x)
```

Parameters

Item	Description
x	[in] A floating-point vector from which to generate Perlin noise.

Return Value

The Perlin noise value within a range between -1 and 1.

Remarks

Perlin noise values change smoothly from one point to another over a space, creating natural looking, randomly generated values. You can use Perlin noise to generate procedural textures for effects like smoke and fire.

Type Description

Name	Template Type	Component Type	Size
x	vector	float	any
ret	scalar	float	1

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	no

Shader Model	Supported
Shader Model 1 (DirectX HLSL)	yes (tx_1_0 only)

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

normalize

Article • 08/19/2020 • 2 minutes to read

Normalizes the specified floating-point vector according to $x / \text{length}(x)$.

```
ret normalize(x)
```

Parameters

Item	Description
x	[in] The specified floating-point vector.

Return Value

The normalized x parameter. If the length of the x parameter is 0, the result is indefinite.

Remarks

The `normalize` HLSL intrinsic function uses the following formula: $x / \text{length}(x)$.

Type Description

Name	Template Type	Component Type	Size
x	<code>vector</code>	<code>float</code>	any
ret	same as input x	<code>float</code>	same dimension(s) as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	yes (vs_1_1 only)

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

pow

Article • 08/19/2020 • 2 minutes to read

Returns the specified value raised to the specified power.

```
ret pow(x, y)
```

Parameters

Item	Description
x	[in] The specified value.
y	[in] The specified power.

Return Value

The x parameter raised to the power of the y parameter.

Remarks

The `pow` HLSL intrinsic function calculates x^y .

X	Y	Result
< 0	any	NAN
> 0	== 0	1
== 0	> 0	0
== 0	< 0	inf
> 0	< 0	$1/X^{-Y}$
== 0	== 0	Depends on the particular graphics processor 0, or 1, or NAN

Type Description

Name	Template Type	Component Type	Size
------	---------------	----------------	------

Name	Template Type	Component Type	Size
x	scalar, vector, or matrix	float	any
y	same as input x	float	same dimension(s) as input x
ret	same as input x	float	same dimension(s) as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	yes (vs_1_1 only)

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

printf function

Article • 10/24/2019 • 2 minutes to read

Submits a custom shader message to the information queue.

Syntax

syntax

```
void printf(  
    string format,  
    argument ...  
) ;
```

Parameters

format

The format string.

argument ...

Optional arguments.

Return value

This function does not return a value.

Remarks

This operation does nothing on devices that do not support it.

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 4 (DirectX HLSL) or later.	yes

See also

[Intrinsic Functions](#)

Process2DQuadTessFactorsAvg function

Article • 03/09/2021 • 2 minutes to read

Generates the corrected tessellation factors for a quad patch.

Syntax

syntax

```
void Process2DQuadTessFactorsAvg(  
    in float4 RawEdgeFactors,  
    in float2 InsideScale,  
    out float4 RoundedEdgeTessFactors,  
    out float2 RoundedInsideTessFactors,  
    out float2 UnroundedInsideTessFactors  
);
```

Parameters

RawEdgeFactors [in]

Type: **float4**

The edge tessellation factors, passed into the tessellator stage.

InsideScale [in]

Type: **float2**

The scale factor applied to the UV tessellation factors computed by the tessellation stage. The allowable range for *InsideScale* is 0.0 to 1.0.

RoundedEdgeTessFactors [out]

Type: **float4**

The rounded edge-tessellation factors calculated by the tessellator stage.

RoundedInsideTessFactors [out]

Type: **float2**

The rounded tessellation factors calculated by the tessellator stage for inside edges.

UnroundedInsideTessFactors [out]

Type: **float2**

The tessellation factors calculated by the tessellator stage for inside edges.

Return value

This function does not return a value.

Remarks

Generates the corrected tessellation factors for a quad patch, computing the inside tessellation factors as the average of the edge tessellation factors. The U and V inside tessellation factors are computed independently using the average of opposing sides of the domain, then are scaled by InsideScale. The result is then rounded based on the partitioning mode, but the unrounded results are available using the UnroundedInsideTessFactors parameter.

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x					

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

Process2DQuadTessFactorsMax function

Article • 03/09/2021 • 2 minutes to read

Generates the corrected tessellation factors for a quad patch.

Syntax

syntax

```
void Process2DQuadTessFactorsMax(  
    in float4 RawEdgeFactors,  
    in float2 InsideScale,  
    out float4 RoundedEdgeTessFactors,  
    out float2 RoundedInsideTessFactors,  
    out float2 UnroundedInsideTessFactors  
);
```

Parameters

RawEdgeFactors [in]

Type: **float4**

The edge tessellation factors, passed into the tessellator stage.

InsideScale [in]

Type: **float2**

The scale factor applied to the UV tessellation factors computed by the tessellation stage. The allowable range for *InsideScale* is 0.0 to 1.0.

RoundedEdgeTessFactors [out]

Type: **float4**

The rounded edge-tessellation factors calculated by the tessellator stage.

RoundedInsideTessFactors [out]

Type: **float2**

The rounded tessellation factors calculated by the tessellator stage for inside edges.

UnroundedInsideTessFactors [out]

Type: `float2`

The tessellation factors calculated by the tessellator stage for inside edges.

Return value

This function does not return a value.

Remarks

Generates the corrected tessellation factors for a quad patch, computing the inside tessellation factors as the maximum of the edge tessellation factors. The U and V inside tessellation factors are computed independently using the maximums of opposing sides of the domain, then are scaled by `InsideScale`. The result is then rounded based on the partitioning mode, but the unrounded results are available using the `UnroundedInsideTessFactors` parameter.

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x					

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

Process2DQuadTessFactorsMin function

Article • 03/09/2021 • 2 minutes to read

Generates the corrected tessellation factors for a quad patch.

Syntax

syntax

```
void Process2DQuadTessFactorsMin(  
    in float4 RawEdgeFactors,  
    in float2 InsideScale,  
    out float4 RoundedEdgeTessFactors,  
    out float2 RoundedInsideTessFactors,  
    out float2 UnroundedInsideTessFactors  
);
```

Parameters

RawEdgeFactors [in]

Type: **float4**

The edge tessellation factors, passed into the tessellator stage.

InsideScale [in]

Type: **float2**

The scale factor applied to the UV tessellation factors computed by the tessellation stage. The allowable range for *InsideScale* is 0.0 to 1.0.

RoundedEdgeTessFactors [out]

Type: **float4**

The rounded edge-tessellation factors calculated by the tessellator stage.

RoundedInsideTessFactors [out]

Type: **float2**

The rounded tessellation factors calculated by the tessellator stage for inside edges.

UnroundedInsideTessFactors [out]

Type: `float2`

The tessellation factors calculated by the tessellator stage for inside edges.

Return value

This function does not return a value.

Remarks

Generates the corrected tessellation factors for a quad patch, computing the inside tessellation factors as the minimum of the edge tessellation factors. The U and V inside tessellation factors are computed independently using the minimums of opposing sides of the domain, then are scaled by `InsideScale`. The result is then rounded based on the partitioning mode, but the unrounded results are available using the `UnroundedInsideTessFactors` parameter.

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x					

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

ProcessIsolineTessFactors function

Article • 10/24/2019 • 2 minutes to read

Generates the rounded tessellation factors for an isoline.

Syntax

syntax

```
void ProcessIsolineTessFactors(  
    in float RawDetailFactor,  
    in float RawDensityFactor,  
    out float RoundedDetailFactor,  
    out float RoundedDensityFactor  
) ;
```

Parameters

RawDetailFactor [in]

Type: **float**

The desired detail factor.

RawDensityFactor [in]

Type: **float**

The desired density factor.

RoundedDetailFactor [out]

Type: **float**

The rounded detail factor clamped to a range that can be used by the tessellator.

RoundedDensityFactor [out]

Type: **float**

The rounded density factor clamped to a rangethat can be used by the tessellator.

Return value

This function does not return a value.

Remarks

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
	x				

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

ProcessQuadTessFactorsAvg function

Article • 03/09/2021 • 2 minutes to read

Generates the corrected tessellation factors for a quad patch.

Syntax

syntax

```
void ProcessQuadTessFactorsAvg(
    in float4 RawEdgeFactors,
    in float InsideScale,
    out float4 RoundedEdgeTessFactors,
    out float2 RoundedInsideTessFactors,
    out float2 UnroundedInsideTessFactors
);
```

Parameters

RawEdgeFactors [in]

Type: **float4**

The edge tessellation factors, passed into the tessellator stage.

InsideScale [in]

Type: **float**

The scale factor applied to the UV tessellation factors computed by the tessellation stage. The allowable range for InsideScale is 0.0 to 1.0.

RoundedEdgeTessFactors [out]

Type: **float4**

The rounded edge-tessellation factors calculated by the tessellator stage.

RoundedInsideTessFactors [out]

Type: **float2**

The rounded tessellation factors calculated by the tessellator stage for inside edges.

UnroundedInsideTessFactors [out]

Type: **float2**

The tessellation factors calculated by the tessellator stage for inside edges.

Return value

This function does not return a value.

Remarks

Generates the corrected tessellation factors for a quad patch, computing the inside tessellation factors as the average of the edge tessellation factors. The inside tess factors will be identical values determined by the average of all four edges scaled by InsideScale. The result is then rounded based on the partitioning mode, but the unrounded results are available using the UnroundedInsideTessFactors parameter.

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
	x				

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

ProcessQuadTessFactorsMax function

Article • 03/09/2021 • 2 minutes to read

Generates the corrected tessellation factors for a quad patch.

Syntax

syntax

```
void ProcessQuadTessFactorsMax(
    in float4 RawEdgeFactors,
    in float InsideScale,
    out float4 RoundedEdgeTessFactors,
    out float2 RoundedInsideTessFactors,
    out float2 UnroundedInsideTessFactors
);
```

Parameters

RawEdgeFactors [in]

Type: **float4**

The edge tessellation factors, passed into the tessellator stage.

InsideScale [in]

Type: **float**

The scale factor applied to the UV tessellation factors computed by the tessellation stage. The allowable range for InsideScale is 0.0 to 1.0.

RoundedEdgeTessFactors [out]

Type: **float4**

The rounded edge-tessellation factors calculated by the tessellator stage.

RoundedInsideTessFactors [out]

Type: **float2**

The rounded tessellation factors calculated by the tessellator stage for inside edges.

UnroundedInsideTessFactors [out]

Type: **float2**

The tessellation factors calculated by the tessellator stage for inside edges.

Return value

This function does not return a value.

Remarks

Generates the corrected tessellation factors for a quad patch, computing the inside tessellation factors as the maximum of the edge tessellation factors. The inside tess factors will be identical values determined by the maximum of all four edges scaled by InsideScale. The result is then rounded based on the partitioning mode, but the unrounded results are available using the *UnroundedInsideTessFactors* parameter.

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
	x				

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

ProcessQuadTessFactorsMin function

Article • 03/09/2021 • 2 minutes to read

Generates the corrected tessellation factors for a quad patch.

Syntax

syntax

```
void ProcessQuadTessFactorsMin(
    in float4 RawEdgeFactors,
    in float InsideScale,
    out float4 RoundedEdgeTessFactors,
    out float2 RoundedInsideTessFactors,
    out float2 UnroundedInsideTessFactors
);
```

Parameters

RawEdgeFactors [in]

Type: **float4**

The edge tessellation factors, passed into the tessellator stage.

InsideScale [in]

Type: **float**

The scale factor applied to the UV tessellation factors computed by the tessellation stage. The allowable range for InsideScale is 0.0 to 1.0.

RoundedEdgeTessFactors [out]

Type: **float4**

The rounded edge-tessellation factors calculated by the tessellator stage.

RoundedInsideTessFactors [out]

Type: **float2**

The rounded tessellation factors calculated by the tessellator stage for inside edges.

UnroundedInsideTessFactors [out]

Type: **float2**

The tessellation factors calculated by the tessellator stage for inside edges.

Return value

This function does not return a value.

Remarks

Generates the corrected tessellation factors for a quad patch, computing the inside tessellation factors as the minimum of the edge tessellation factors. The inside tess factors will be identical values determined by the minimum of all four edges scaled by InsideScale. The result is then rounded based on the partitioning mode, but the unrounded results are available using the UnroundedInsideTessFactors parameter.

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
	x				

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

ProcessTriTessFactorsAvg function

Article • 03/09/2021 • 2 minutes to read

Generates the corrected tessellation factors for a tri patch.

Syntax

syntax

```
void ProcessTriTessFactorsAvg(  
    in float3 RawEdgeFactors,  
    in float InsideScale,  
    out float3 RoundedEdgeTessFactors,  
    out float RoundedInsideTessFactor,  
    out float UnroundedInsideTessFactor  
) ;
```

Parameters

RawEdgeFactors [in]

Type: **float3**

The edge tessellation factors, passed into the tessellator stage.

InsideScale [in]

Type: **float**

The scale factor applied to the UV tessellation factors computed by the tessellation stage. The allowable range for InsideScale is 0.0 to 1.0.

RoundedEdgeTessFactors [out]

Type: **float3**

The rounded edge-tessellation factors calculated by the tessellator stage.

RoundedInsideTessFactor [out]

Type: **float**

The tessellation factors calculated by the tessellator stage, and rounded.

UnroundedInsideTessFactor [out]

Type: **float**

The original, unrounded, UV tessellation factors computed by the tessellation stage.

Return value

This function does not return a value.

Remarks

Generates the corrected tessellation factors for a tri patch, computing the inside tessellation factor as the average of the edge tessellation factors, which is then scaled by InsideScale. The result is then rounded based on the partitioning mode, but the unrounded results are available using the UnroundedInsideTessFactor parameter.

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x					

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

ProcessTriTessFactorsMax function

Article • 03/09/2021 • 2 minutes to read

Generates the corrected tessellation factors for a tri patch.

Syntax

syntax

```
void ProcessTriTessFactorsMax(  
    in float3 RawEdgeFactors,  
    in float InsideScale,  
    out float3 RoundedEdgeTessFactors,  
    out float RoundedInsideTessFactor,  
    out float UnroundedInsideTessFactor  
);
```

Parameters

RawEdgeFactors [in]

Type: **float3**

The edge tessellation factors, passed into the tessellator stage.

InsideScale [in]

Type: **float**

The scale factor applied to the UV tessellation factors computed by the tessellation stage. The allowable range for InsideScale is 0.0 to 1.0.

RoundedEdgeTessFactors [out]

Type: **float3**

The rounded edge-tessellation factors calculated by the tessellator stage.

RoundedInsideTessFactor [out]

Type: **float**

The tessellation factors calculated by the tessellator stage, and rounded.

UnroundedInsideTessFactor [out]

Type: **float**

The original, unrounded, UV tessellation factors computed by the tessellation stage.

Return value

This function does not return a value.

Remarks

Generates the corrected tessellation factors for a tri patch, computing the inside tessellation factor as the maximum of the edge tessellation factors, which is then scaled by InsideScale. The result is then rounded based on the partitioning mode, but the unrounded results are available using the UnroundedInsideTessFactor parameter.

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
	x				

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

ProcessTriTessFactorsMin function

Article • 03/09/2021 • 2 minutes to read

Generates the corrected tessellation factors for a tri patch.

Syntax

syntax

```
void ProcessTriTessFactorsMin(  
    in float3 RawEdgeFactors,  
    in float InsideScale,  
    out float3 RoundedEdgeTessFactors,  
    out float RoundedInsideTessFactors,  
    out float UnroundedInsideTessFactors  
);
```

Parameters

RawEdgeFactors [in]

Type: **float3**

The edge tessellation factors, passed into the tessellator stage.

InsideScale [in]

Type: **float**

The scale factor applied to the UV tessellation factors computed by the tessellation stage. The allowable range for InsideScale is 0.0 to 1.0.

RoundedEdgeTessFactors [out]

Type: **float3**

The rounded edge-tessellation factors calculated by the tessellator stage.

RoundedInsideTessFactors [out]

Type: **float**

The rounded tessellation factors calculated by the tessellator stage for inside edges.

UnroundedInsideTessFactors [out]

Type: **float**

The tessellation factors calculated by the tessellator stage for inside edges.

Return value

This function does not return a value.

Remarks

Generates the corrected tessellation factors for a tri patch, computing the inside tessellation factor as the minimum of the edge tessellation factors, which is then scaled by InsideScale. The result is then rounded based on the partitioning mode, but the unrounded results are available using the UnroundedInsideTessFactor parameter.

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x					

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

radians

Article • 08/19/2020 • 2 minutes to read

Converts the specified value from degrees to radians.

```
ret radians(x)
```

Parameters

Item	Description
x	[in] The specified value.

Return Value

The x parameter converted from degrees to radians.

Type Description

Name	Template Type	Component Type	Size
x	scalar, vector, or matrix	float	any
ret	same as input x	float	same dimension(s) as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 1 (DirectX HLSL) and higher shader models	yes

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

rcp

Article • 08/19/2020 • 2 minutes to read

Calculates a fast, approximate, per-component reciprocal.

```
ret rcp(x)
```

Parameters

x

[in] The input value.

Return Value

The reciprocal of the *x* parameter.

Type Description

Name	Template Type	Component Type	Size
<i>x</i>	scalar, vector, or matrix	float or double	any
<i>ret</i>	same as input <i>x</i>	float or double	same dimension(s) as input <i>x</i>

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

reflect

Article • 08/19/2020 • 2 minutes to read

Returns a reflection vector using an incident ray and a surface normal.

```
ret reflect(i, n)
```

Parameters

Item	Description
<i>i</i>	[in] A floating-point, incident vector.
<i>n</i>	[in] A floating-point, normal vector.

Return Value

A floating-point, reflection vector.

Remarks

This function calculates the reflection vector using the following formula: $v = i - 2 * n * \text{dot}(i n)$.

Type Description

Name	Template Type	Component Type	Size
<i>i</i>	vector	float	any
<i>n</i>	vector	float	same dimension(s) as input <i>i</i>
<i>ret</i>	vector	float	same dimension(s) as input <i>i</i>

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
1.1	Yes

Shader Model	Supported
Shader Model 1 (DirectX HLSL) and higher shader models	yes

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

refract

Article • 08/19/2020 • 2 minutes to read

Returns a refraction vector using an entering ray, a surface normal, and a refraction index.

```
ret refract(i, n, ?)
```

Parameters

Item	Description
<i>i</i>	[in] A floating-point, ray direction vector.
<i>n</i>	[in] A floating-point, surface normal vector.
?	[in] A floating-point, refraction index scalar.

Return Value

A floating-point, refraction vector. If the angle between the entering ray *i* and the surface normal *n* is too great for a given refraction index ?, the return value is (0,0,0).

Type Description

Name	Template Type	Component Type	Size
<i>i</i>	vector	float	any
<i>n</i>	vector	float	same dimension(s) as input <i>i</i>
?	scalar	float	1
refraction vector	vector	float	same dimension(s) as input <i>i</i>

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	yes (vs_1_1 only)

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

reversebits function

Article • 10/24/2019 • 2 minutes to read

Reverses the order of the bits, per component.

Syntax

syntax

```
uint reversebits(  
    in uint value  
) ;
```

Parameters

value [in]

Type: **uint**

The input value.

Return value

Type: **uint**

The input value, with the bit order reversed.

Remarks

The following overloaded versions are also available:

syntax

```
uint2 reversebits(uint2 value);  
uint3 reversebits(uint3 value);  
uint4 reversebits(uint4 value);
```

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5 and higher shader models	yes

This function is supported in the following types of shaders:

Vertex	Hull	Domain	Geometry	Pixel	Compute
x	x	x	x	x	x

See also

[Intrinsic Functions](#)

[Shader Model 5](#)

round

Article • 03/15/2021 • 2 minutes to read

Rounds the specified value to the nearest integer. Halfway cases are rounded to the nearest even.

```
ret round(x)
```

Parameters

Item	Description
x	[in] The specified value.

Return Value

The x parameter, rounded to the nearest integer within a floating-point type.

Type Description

Name	Template Type	Component Type	Size
x	scalar, vector, or matrix	float	any
ret	same as input x	float	same dimension(s) as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	yes (vs_1_1 only)

Requirements

Requirement	Value

Requirement	Value
Header	Corecrt_math.h

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

rsqrt

Article • 08/19/2020 • 2 minutes to read

Returns the reciprocal of the square root of the specified value.

```
ret rsqrt(x)
```

Parameters

Item	Description
x	[in] The specified value.

Return Value

The reciprocal of the square root of the *x* parameter.

Remarks

This function uses the following formula: $1 / \sqrt{x}$.

Type Description

Name	Template Type	Component Type	Size
x	scalar , vector , or matrix	float	any
ret	same as input x	float	same dimension(s) as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	yes (vs_1_1 only)

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

saturate (HLSL reference)

Article • 08/19/2020 • 2 minutes to read

Clamps the specified value within the range of 0 to 1.

```
ret saturate(x)
```

Parameters

Item	Description
x	[in] The specified value.

Return Value

The x parameter, clamped within the range of 0 to 1.

Type Description

Name	Template Type	Component Type	Size
x	scalar, vector, or matrix	float	any
ret	same as input x	float	same dimension(s) as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 1 (DirectX HLSL) and higher shader models	yes

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

sign

Article • 08/19/2020 • 2 minutes to read

Returns the sign of x .

```
ret sign(x)
```

Parameters

Item	Description
x	[in] The input value.

Return Value

Returns -1 if x is less than zero; 0 if x equals zero; and 1 if x is greater than zero.

Type Description

Name	Template Type	Component Type	Size
x	scalar, vector, or matrix	float, int	any
ret	same as input x	int	same dimension(s) as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	yes (vs_1_1 and ps_1_4)

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

sin

Article • 08/19/2020 • 2 minutes to read

Returns the sine of the specified value.

```
ret sin(x)
```

Parameters

Item	Description
x	[in] The specified value, in radians.

Return Value

The sine of the *x* parameter.

Type Description

Name	Template Type	Component Type	Size
x	scalar, vector, or matrix	float	any
ret	same as input x	float	same dimension(s) as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	yes (vs_1_1 only)

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

sincos

Article • 08/19/2020 • 2 minutes to read

Returns the sine and cosine of x.

```
sincos(x, out s, out c)
```

Parameters

Item	Description
x	[in] The specified value, in radians.
s	[out] Returns the sine of x.
c	[out] Returns the cosine of x.

Return Value

None.

Type Description

Name	Template Type	Component Type	Size
x	scalar, vector, or matrix	float	any
s	same as input x	float	same dimension(s) as input x
c	same as input x	float	same dimension(s) as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	yes (vs_1_1 only)

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

sinh

Article • 08/19/2020 • 2 minutes to read

Returns the hyperbolic sine of the specified value.

```
ret sinh(x)
```

Parameters

Item	Description
x	[in] The specified value, in radians.

Return Value

The hyperbolic sine of the *x* parameter.

Type Description

Name	Template Type	Component Type	Size
x	scalar, vector, or matrix	float	any
ret	same as input x	float	same dimension(s) as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	yes (vs_1_1 only)

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

smoothstep

Article • 08/19/2020 • 2 minutes to read

Returns a smooth Hermite interpolation between 0 and 1, if x is in the range $[min, max]$.

```
ret smoothstep(min, max, x)
```

Parameters

Item	Description
<i>min</i>	[in] The minimum range of the x parameter.
<i>max</i>	[in] The maximum range of the x parameter.
<i>x</i>	[in] The specified value to be interpolated.

Return Value

Returns 0 if x is less than *min*; 1 if x is greater than *max*; otherwise, a value between 0 and 1 if x is in the range $[min, max]$.

Remarks

Use the **smoothstep** HLSL intrinsic function to create a smooth transition between two values. For example, you can use this function to blend two colors smoothly.

Type Description

Name	Template Type	Component Type	Size
<i>x</i>	scalar , vector , or matrix	float	any
<i>min</i>	same as input <i>x</i>	float	same dimension(s) as input <i>x</i>
<i>max</i>	same as input <i>x</i>	float	same dimension(s) as input <i>x</i>
<i>ret</i>	same as input <i>x</i>	float	same dimension(s) as input <i>x</i>

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	yes (vs_1_1 only)

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

sqrt

Article • 08/19/2020 • 2 minutes to read

Returns the square root of the specified floating-point value, per component.

```
ret sqrt(x)
```

Parameters

Item	Description
x	[in] The specified floating-point value.

Return Value

The square root of the x parameter, per component.

Type Description

Name	Template Type	Component Type	Size
x	scalar, vector, or matrix	float	any
ret	same as input x	float	same dimension(s) as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	yes (vs_1_1 only)

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

step

Article • 08/19/2020 • 2 minutes to read

Compares two values, returning 0 or 1 based on which value is greater.

```
ret step(y, x)
```

Parameters

Item	Description
y	[in] The first floating-point value to compare.
x	[in] The second floating-point value to compare.

Return Value

1 if the x parameter is greater than or equal to the y parameter; otherwise, 0.

Remarks

This function uses the following formula: $(x \geq y) ? 1 : 0$. The function returns either 0 or 1 depending on whether the x parameter is greater than the y parameter. To compute a smooth interpolation between 0 and 1, use the [smoothstep](#) HLSL intrinsic function.

Type Description

Name	Template Type	Component Type	Size
y	scalar , vector , or matrix	float	any
x	same as input y	float	same dimension(s) as input y
ret	same as input y	float	same dimension(s) as input y

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	yes (vs_1_1 and ps_1_4)

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

tan

Article • 08/19/2020 • 2 minutes to read

Returns the tangent of the specified value.

```
ret tan(x)
```

Parameters

Item	Description
x	[in] The specified value, in radians.

Return Value

The tangent of the x parameter.

Type Description

Name	Template Type	Component Type	Size
x	scalar, vector, or matrix	float	any
ret	same as input x	float	same dimension(s) as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	yes (vs_1_1 only)

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

tanh

Article • 08/19/2020 • 2 minutes to read

Returns the hyperbolic tangent of the specified value.

```
ret tanh(x)
```

Parameters

Item	Description
x	[in] The specified value, in radians.

Return Value

The hyperbolic tangent of the *x* parameter.

Type Description

Name	Template Type	Component Type	Size
x	scalar, vector, or matrix	float	any
ret	same as input x	float	same dimension(s) as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 2 (DirectX HLSL) and higher shader models	yes
Shader Model 1 (DirectX HLSL)	yes (vs_1_1 only)

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

tex1D (HLSL reference)

Article • 08/19/2020 • 2 minutes to read

Samples a 1D texture.

```
ret tex1D(s, t)
```

Parameters

Item	Description
s	[in] The sampler state.
t	[in] The texture coordinate.

Return Value

The value of the texture data.

Type Description

Name	In/Out	Template Type	Component Type	Size
s	in	object	sampler1D	1
t	in	scalar	float	1
ret	out	vector	float	4

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 4	yes (pixel shader only), but you must use the legacy compile option when compiling.
Shader Model 3 (DirectX HLSL)	yes (pixel shader only)

Shader Model	Supported
Shader Model 2 (DirectX HLSL)	yes (pixel shader only)
Shader Model 1 (DirectX HLSL)	yes (pixel shader only)

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

tex1D (HLSL reference) - Select the mip level

Article • 03/15/2021 • 2 minutes to read

Samples a 1D texture using a gradient to select the mip level.

```
ret tex1D(s, t, ddx, ddy)
```

Parameters

Item	Description
s	[in] The sampler state.
t	[in] The texture coordinate.
ddx	[in] Rate of change of the surface geometry in the x direction.
ddy	[in] Rate of change of the surface geometry in the y direction.

Return Value

The value of the texture data.

Type Description

Name	In/Out	Template Type	Component Type	Size
s	in	object	sampler1D	1
t	in	vector	float	1
ddx	in	vector	float	1
ddy	in	vector	float	1
ret	out	vector	float	4

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 4	yes (pixel shader only)
Shader Model 3 (DirectX HLSL)	yes (pixel shader only)
Shader Model 2 (DirectX HLSL)	yes (pixel shader only)
Shader Model 1 (DirectX HLSL)	no

1. Significant code reordering is done to move gradient computations outside of flow control.
2. If the D3DPSHADERCAPS2_0 cap is set with D3DD3DPSHADERCAPS2_0_GRADIENTINSTRUCTIONS, the compiler maps this function to texIdd.

Remarks

When flow control is present in a shader, the result of a gradient calculation requested inside a given branch path is ambiguous when adjacent pixels may go down separate flow control paths. Therefore, it is deemed illegal to use any pixel shader operation that requests a gradient calculation to occur at a location that is inside a flow control construct which could vary across pixels for a given primitive being rasterized. If either side of an if statement with the branch attribute uses a gradient function a compiler error may be generated. See [if Statement \(DirectX HLSL\)](#).

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

tex1Dbias

Article • 08/19/2020 • 2 minutes to read

Samples a 1D texture after biasing the mip level by t.w.

```
ret tex1Dbias(s, t)
```

Parameters

Item	Description
s	[in] The sampler state.
t	[in] The texture coordinate.

Return Value

The value of the texture data.

Type Description

Name	In/Out	Template Type	Component Type	Size
s	in	object	sampler1D	1
t	in	vector	float	4
ret	out	vector	float	4

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 4	yes (pixel shader only)
Shader Model 3 (DirectX HLSL)	yes (pixel shader only)
Shader Model 2 (DirectX HLSL)	yes (pixel shader only)

Shader Model	Supported
Shader Model 1 (DirectX HLSL)	no

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

tex1Dgrad

Article • 03/09/2021 • 2 minutes to read

Samples a 1D texture using a gradient to select the mip level.

```
ret tex1Dgrad(s, t, ddx, ddy)
```

Parameters

Item	Description
s	[in] The sampler state.
t	[in] The texture coordinate.
ddx	[in] Rate of change of the surface geometry in the x direction.
ddy	[in] Rate of change of the surface geometry in the y direction.

Return Value

The value of the texture data.

Type Description

Name	In/Out	Template Type	Component Type	Size
s	in	object	sampler1D	1
t	in	vector	float	1
ddx	in	vector	float	1
ddy	in	vector	float	1
ret	out	vector	float	4

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 4	yes (pixel shader only)
Shader Model 3 (DirectX HLSL)	yes (pixel shader only)
Shader Model 2 (DirectX HLSL)	yes (pixel shader only)
Shader Model 1 (DirectX HLSL)	no

1. Significant code reordering is done to move gradient computations outside of flow control.
2. If the D3DPSHADERCAPS2_0 cap is set with D3DD3DPSHADERCAPS2_0_GRADIENTINSTRUCTIONS, the compiler maps this function to texldd.

Remarks

When flow control is present in a shader, the result of a gradient calculation requested inside a given branch path is ambiguous when adjacent pixels may go down separate flow control paths. Therefore, it is deemed illegal to use any pixel shader operation that requests a gradient calculation to occur at a location that is inside a flow control construct which could vary across pixels for a given primitive being rasterized. If either side of an if statement with the branch attribute uses a gradient function a compiler error may be generated. See [if Statement \(DirectX HLSL\)](#).

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

tex1Dlod

Article • 08/19/2020 • 2 minutes to read

Samples a 1D texture with mipmaps. The mipmap LOD is specified in t.w.

```
ret tex1Dlod(s, t)
```

Parameters

Item	Description
s	[in] The sampler state.
t	[in] The texture coordinate.

Return Value

The value of the texture data.

Type Description

Name	In/Out	Template Type	Component Type	Size
s	in	object	sampler1D	1
t	in	vector	float	4
ret	out	vector	float	4

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 4	yes (pixel shader only)
Shader Model 3 (DirectX HLSL)	yes (pixel shader only)
Shader Model 2 (DirectX HLSL)	no

Shader Model	Supported
Shader Model 1 (DirectX HLSL)	no

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

tex1Dproj

Article • 08/19/2020 • 2 minutes to read

Samples a 1D texture using a projective divide; the texture coordinate is divided by `t.w` before the lookup takes place.

```
ret tex1Dproj(s, t)
```

Parameters

Item	Description
<code>s</code>	[in] The sampler state.
<code>t</code>	[in] The texture coordinate.

Return Value

The value of the texture data.

Type Description

Name	In/Out	Template Type	Component Type	Size
<code>s</code>	in	object	sampler1D	1
<code>t</code>	in	vector	float	4
<code>ret</code>	out	vector	float	4

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 4	yes (pixel shader only)
Shader Model 3 (DirectX HLSL)	yes (pixel shader only)
Shader Model 2 (DirectX HLSL)	yes (pixel shader only)

Shader Model	Supported
Shader Model 1 (DirectX HLSL)	no

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

tex2D (HLSL reference)

Article • 08/19/2020 • 2 minutes to read

Samples a 2D texture.

```
ret tex2D(s, t)
```

Parameters

Item	Description
s	[in] The sampler state.
t	[in] The texture coordinate.

Return Value

The value of the texture data.

Type Description

Name	In/Out	Template Type	Component Type	Size
s	in	object	sampler2D	1
t	in	vector	float	2
ret	out	vector	float	4

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 4	yes (pixel shader only), but you must use the legacy compile option when compiling.
Shader Model 3 (DirectX HLSL)	yes (pixel shader only)

Shader Model	Supported
Shader Model 2 (DirectX HLSL)	yes (pixel shader only)
Shader Model 1 (DirectX HLSL)	yes (pixel shader only)

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

tex2D (HLSL reference) - Select the mip level

Article • 03/15/2021 • 2 minutes to read

Samples a 2D texture using a gradient to select the mip level.

```
ret tex2D(s, t, ddx, ddy)
```

Parameters

Item	Description
s	[in] The sampler state.
t	[in] The texture coordinates.
ddx	[in] Rate of change of the surface geometry in the x direction.
ddy	[in] Rate of change of the surface geometry in the y direction.

Return Value

The value of the texture data.

Type Description

Name	In/Out	Template Type	Component Type	Size
s	in	object	sampler2D	1
t	in	vector	float	2
ddx	in	vector	float	2
ddy	in	vector	float	2
ret	out	vector	float	4

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 4	yes (pixel shader only)
Shader Model 3 (DirectX HLSL)	yes (pixel shader only)
Shader Model 2 (DirectX HLSL)	yes (pixel shader only)
Shader Model 1 (DirectX HLSL)	no

1. Significant code reordering is done to move gradient computations outside of flow control.
2. If the D3DPSHADERCAPS2_0 cap is set with D3DD3DPSHADERCAPS2_0_GRADIENTINSTRUCTIONS, the compiler maps this function to texIdd.

Remarks

When flow control is present in a shader, the result of a gradient calculation requested inside a given branch path is ambiguous when adjacent pixels may go down separate flow control paths. Therefore, it is deemed illegal to use any pixel shader operation that requests a gradient calculation to occur at a location that is inside a flow control construct which could vary across pixels for a given primitive being rasterized. If either side of an if statement with the branch attribute uses a gradient function a compiler error may be generated. See [if Statement \(DirectX HLSL\)](#).

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

tex2Dbias

Article • 08/19/2020 • 2 minutes to read

Samples a 2D texture after biasing the mip level by t.w.

```
ret tex2Dbias(s, t)
```

Parameters

Item	Description
s	[in] The sampler state.
t	[in] The texture coordinate.

Return Value

The value of the texture data.

Type Description

Name	In/Out	Template Type	Component Type	Size
s	in	object	sampler2D	1
t	in	vector	float	4
ret	out	vector	float	4

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 4	yes (pixel shader only)
Shader Model 3 (DirectX HLSL)	yes (pixel shader only)
Shader Model 2 (DirectX HLSL)	yes (pixel shader only)

Shader Model	Supported
Shader Model 1 (DirectX HLSL)	no

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

tex2Dgrad

Article • 03/09/2021 • 2 minutes to read

Samples a 2D texture using a gradient to select the mip level.

```
ret tex2Dgrad(s, t, ddx, ddy)
```

Parameters

Item	Description
s	[in] The sampler state.
t	[in] The texture coordinates.
ddx	[in] Rate of change of the surface geometry in the x direction.
ddy	[in] Rate of change of the surface geometry in the y direction.

Return Value

The value of the texture data.

Type Description

Name	In/Out	Template Type	Component Type	Size
s	in	object	sampler2D	1
t	in	vector	float	2
ddx	in	vector	float	2
ddy	in	vector	float	2
ret	out	vector	float	4

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 4	yes (pixel shader only)
Shader Model 3 (DirectX HLSL)	yes (pixel shader only)
Shader Model 2 (DirectX HLSL)	yes (pixel shader only)
Shader Model 1 (DirectX HLSL)	no

1. Significant code reordering is done to move gradient computations outside of flow control.
2. If the D3DPSHADERCAPS2_0 cap is set with D3DD3DPSHADERCAPS2_0_GRADIENTINSTRUCTIONS, the compiler maps this function to texldd.

Remarks

When flow control is present in a shader, the result of a gradient calculation requested inside a given branch path is ambiguous when adjacent pixels may go down separate flow control paths. Therefore, it is deemed illegal to use any pixel shader operation that requests a gradient calculation to occur at a location that is inside a flow control construct which could vary across pixels for a given primitive being rasterized. If either side of an if statement with the branch attribute uses a gradient function a compiler error may be generated. See [if Statement \(DirectX HLSL\)](#).

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

tex2Dlod

Article • 08/19/2020 • 2 minutes to read

Samples a 2D texture with mipmaps. The mipmap LOD is specified in t.w.

```
ret tex2Dlod(s, t)
```

Parameters

Item	Description
s	[in] The sampler state.
t	[in] The texture coordinate.

Return Value

The value of the texture data.

Type Description

Name	In/Out	Template Type	Component Type	Size
s	in	object	sampler2D	1
t	in	vector	float	4
ret	out	vector	float	4

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 3 (DirectX HLSL) and higher shader models	yes
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Remarks

Starting with Direct3D 10, you can use new HLSL syntax to access textures and other resources. You can replace intrinsic-style texture lookup functions, such as `tex2Dlod`, with a more object-oriented style. In this object-oriented style, textures are decoupled from samplers and have methods for loading and sampling.

To sample a 2D texture, instead of using `tex2Dlod` as in this code:

```
sampler S;  
...  
color = tex2Dlod(S, Location);
```

Use the [SampleLevel](#) method of a [Texture Object](#) as in this code:

```
Texture2D MyTexture;  
SamplerState MySampler;  
...  
color = MyTexture.SampleLevel(MySampler, Location, LOD);
```

To use the intrinsic-style texture lookup functions, such as `tex2Dlod`, with [shader model 4](#) and higher, use [D3DCOMPILE_ENABLE_BACKWARDS_COMPATIBILITY](#) to compile. However, if you want to target shader model 4 and higher (even `*_4_0_level_9_*`) with newer object-oriented style code, migrate to the newer HLSL syntax.

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

tex2Dproj

Article • 08/19/2020 • 2 minutes to read

Samples a 2D texture using a projective divide; the texture coordinate is divided by `t.w` before the lookup takes place.

```
ret tex2Dproj(s, t)
```

Parameters

Item	Description
<code>s</code>	[in] The sampler state.
<code>t</code>	[in] The texture coordinate.

Return Value

The value of the texture data.

Type Description

Name	In/Out	Template Type	Component Type	Size
<code>s</code>	in	object	sampler2D	1
<code>t</code>	in	vector	float	4
<code>ret</code>	out	vector	float	4

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 4	yes (pixel shader only)
Shader Model 3 (DirectX HLSL)	yes (pixel shader only)
Shader Model 2 (DirectX HLSL)	yes (pixel shader only)

Shader Model	Supported
Shader Model 1 (DirectX HLSL)	no

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

tex3D (HLSL reference)

Article • 08/19/2020 • 2 minutes to read

Samples a 3D texture.

```
ret tex3D(s, t)
```

Parameters

Item	Description
s	[in] The sampler state.
t	[in] The texture coordinate.

Return Value

The value of the texture data.

Type Description

Name	In/Out	Template Type	Component Type	Size
s	in	object	sampler3D	1
t	in	vector	float	3
ret	out	vector	float	4

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 4	yes (pixel shader only), but you must use the legacy compile option when compiling.
Shader Model 3 (DirectX HLSL)	yes (pixel shader only)

Shader Model	Supported
Shader Model 2 (DirectX HLSL)	yes (pixel shader only)
Shader Model 1 (DirectX HLSL)	yes (pixel shader only)

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

tex3D (HLSL reference) - Select the mip level

Article • 03/15/2021 • 2 minutes to read

Samples a 3D texture using a gradient to select the mip level.

```
ret tex3D(s, t, ddx, ddy)
```

Parameters

Item	Description
s	[in] The sampler state.
t	[in] The texture coordinate.
ddx	[in] Rate of change of the surface geometry in the x direction.
ddy	[in] Rate of change of the surface geometry in the y direction.

Return Value

The value of the texture data.

Type Description

Name	In/Out	Template Type	Component Type	Size
s	in	object	sampler3D	1
t	in	vector	float	3
ddx	in	vector	float	3
ddy	in	vector	float	3
ret	out	vector	float	4

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 4	yes (pixel shader only)
Shader Model 3 (DirectX HLSL)	yes (pixel shader only)
Shader Model 2 (DirectX HLSL)	yes (pixel shader only)
Shader Model 1 (DirectX HLSL)	no

1. Significant code reordering is done to move gradient computations outside of flow control.
2. If the D3DPSHADERCAPS2_0 cap is set with D3DD3DPSHADERCAPS2_0_GRADIENTINSTRUCTIONS, the compiler maps this function to texIdd.

Remarks

When flow control is present in a shader, the result of a gradient calculation requested inside a given branch path is ambiguous when adjacent pixels may go down separate flow control paths. Therefore, it is deemed illegal to use any pixel shader operation that requests a gradient calculation to occur at a location that is inside a flow control construct which could vary across pixels for a given primitive being rasterized. If either side of an if statement with the branch attribute uses a gradient function a compiler error may be generated. See [if Statement \(DirectX HLSL\)](#).

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

tex3Dbias

Article • 08/19/2020 • 2 minutes to read

Samples a 3D texture after biasing the mip level by t.w.

```
ret tex3Dbias(s, t)
```

Parameters

Item	Description
s	[in] The sampler state.
t	[in] The texture coordinate.

Return Value

The value of the texture data.

Type Description

Name	In/Out	Template Type	Component Type	Size
s	in	object	sampler3D	1
t	in	vector	float	4
ret	out	vector	float	4

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 4	yes (pixel shader only)
Shader Model 3 (DirectX HLSL)	yes (pixel shader only)
Shader Model 2 (DirectX HLSL)	yes (pixel shader only)

Shader Model	Supported
Shader Model 1 (DirectX HLSL)	no

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

tex3Dgrad

Article • 03/09/2021 • 2 minutes to read

Samples a 3D texture using a gradient to select the mip level.

```
ret tex3Dgrad(s, t, ddx, ddy)
```

Parameters

Item	Description
s	[in] The sampler state.
t	[in] The texture coordinate.
ddx	[in] Rate of change of the surface geometry in the x direction.
ddy	[in] Rate of change of the surface geometry in the y direction.

Return Value

The value of the texture data.

Type Description

Name	In/Out	Template Type	Component Type	Size
s	in	object	sampler3D	1
t	in	vector	float	3
ddx	in	vector	float	3
ddy	in	vector	float	3
ret	out	vector	float	4

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 4	yes (pixel shader only)
Shader Model 3 (DirectX HLSL)	yes (pixel shader only)
Shader Model 2 (DirectX HLSL)	yes (pixel shader only)
Shader Model 1 (DirectX HLSL)	no

1. Significant code reordering is done to move gradient computations outside of flow control.
2. If the D3DPSHADERCAPS2_0 cap is set with D3DD3DPSHADERCAPS2_0_GRADIENTINSTRUCTIONS, the compiler maps this function to texldd.

Remarks

When flow control is present in a shader, the result of a gradient calculation requested inside a given branch path is ambiguous when adjacent pixels may go down separate flow control paths. Therefore, it is deemed illegal to use any pixel shader operation that requests a gradient calculation to occur at a location that is inside a flow control construct which could vary across pixels for a given primitive being rasterized. If either side of an if statement with the branch attribute uses a gradient function a compiler error may be generated. See [if Statement \(DirectX HLSL\)](#).

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

tex3Dlod

Article • 08/19/2020 • 2 minutes to read

Samples a 3D texture with mipmaps. The mipmap LOD is specified in `t.w`.

```
ret tex3Dlod(s, t)
```

Parameters

Item	Description
<code>s</code>	[in] The sampler state.
<code>t</code>	[in] The texture coordinate.

Return Value

The value of the texture data.

Type Description

Name	In/Out	Template Type	Component Type	Size
<code>s</code>	in	<code>object</code>	<code>sampler3D</code>	1
<code>t</code>	in	<code>vector</code>	<code>float</code>	4
<code>ret</code>	out	<code>vector</code>	<code>float</code>	4

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 4	yes (pixel shader only)
Shader Model 3 (DirectX HLSL)	yes (pixel shader only)
Shader Model 2 (DirectX HLSL)	no

Shader Model	Supported
Shader Model 1 (DirectX HLSL)	no

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

tex3Dproj

Article • 08/19/2020 • 2 minutes to read

Samples a 3D texture using a projective divide; the texture coordinate is divided by `t.w` before the lookup takes place.

```
ret tex3Dproj(s, t)
```

Parameters

Item	Description
<code>s</code>	[in] The sampler state.
<code>t</code>	[in] The texture coordinate.

Return Value

The value of the texture data.

Type Description

Name	In/Out	Template Type	Component Type	Size
<code>s</code>	in	object	sampler3D	1
<code>t</code>	in	vector	float	4
<code>ret</code>	out	vector	float	4

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 4	yes (pixel shader only)
Shader Model 3 (DirectX HLSL)	yes (pixel shader only)
Shader Model 2 (DirectX HLSL)	yes (pixel shader only)

Shader Model	Supported
Shader Model 1 (DirectX HLSL)	no

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

texCUBE (HLSL reference)

Article • 08/19/2020 • 2 minutes to read

Samples a cube texture.

```
ret texCUBE(s, t)
```

Parameters

Item	Description
s	[in] The sampler state.
t	[in] The texture coordinate.

Return Value

The value of the texture data.

Type Description

Name	In/Out	Template Type	Component Type	Size
s	in	object	samplerCUBE	1
t	in	vector	float	3
ret	out	vector	float	4

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 4	yes (pixel shader only)
Shader Model 3 (DirectX HLSL)	yes (pixel shader only)
Shader Model 2 (DirectX HLSL)	yes (pixel shader only)

Shader Model	Supported
Shader Model 1 (DirectX HLSL)	yes (pixel shader only)

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

texCUBE (HLSL reference) - Select the mip level

Article • 03/15/2021 • 2 minutes to read

Samples a cube texture using a gradient to select the mip level.

```
ret texCUBE(s, t, ddx, ddy)
```

Parameters

Item	Description
s	[in] The sampler state.
t	[in] The texture coordinate.
ddx	[in] Rate of change of the surface geometry in the x direction.
ddy	[in] Rate of change of the surface geometry in the y direction.

Return Value

The value of the texture data.

Type Description

Name	In/Out	Template Type	Component Type	Size
s	in	object	samplerCUBE	1
t	in	vector	float	3
ddx	in	vector	float	3
ddy	in	vector	float	3
ret	out	vector	float	4

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 4	yes (pixel shader only)
Shader Model 3 (DirectX HLSL)	yes (pixel shader only)
Shader Model 2 (DirectX HLSL)	yes (pixel shader only)
Shader Model 1 (DirectX HLSL)	no

1. Significant code reordering is done to move gradient computations outside of flow control.
2. If the D3DPSHADERCAPS2_0 cap is set with D3DD3DPSHADERCAPS2_0_GRADIENTINSTRUCTIONS, the compiler maps this function to texIdd.

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

texCUBEbias

Article • 08/19/2020 • 2 minutes to read

Samples a cube texture after biasing the mip level by t.w.

```
ret texCUBEbias(s, t)
```

Parameters

Item	Description
s	[in] The sampler state.
t	[in] The texture coordinate.

Return Value

The value of the texture data.

Type Description

Name	In/Out	Template Type	Component Type	Size
s	in	object	samplerCUBE	1
t	in	vector	float	4
ret	out	vector	float	4

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 4	yes (pixel shader only)
Shader Model 3 (DirectX HLSL)	yes (pixel shader only)
Shader Model 2 (DirectX HLSL)	yes (pixel shader only)

Shader Model	Supported
Shader Model 1 (DirectX HLSL)	no

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

texCUBEgrad

Article • 03/09/2021 • 2 minutes to read

Samples a cube texture using a gradient to select the mip level.

```
ret texCUBEgrad(s, t, ddx, ddy)
```

Parameters

Item	Description
s	[in] The sampler state.
t	[in] The texture coordinate.
ddx	[in] Rate of change of the surface geometry in the x direction.
ddy	[in] Rate of change of the surface geometry in the y direction.

Return Value

The value of the texture data.

Type Description

Name	In/Out	Template Type	Component Type	Size
s	in	object	samplerCUBE	1
t	in	vector	float	3
ddx	in	vector	float	3
ddy	in	vector	float	3
ret	out	vector	float	4

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 4	yes (pixel shader only)
Shader Model 3 (DirectX HLSL)	yes (pixel shader only)
Shader Model 2 (DirectX HLSL)	yes (pixel shader only)
Shader Model 1 (DirectX HLSL)	no

1. Significant code reordering is done to move gradient computations outside of flow control.
2. If the D3DPSHADERCAPS2_0 cap is set with D3DD3DPSHADERCAPS2_0_GRADIENTINSTRUCTIONS, the compiler maps this function to texldd.

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

texCUBElod

Article • 08/19/2020 • 2 minutes to read

Samples a cube texture with mipmaps. The mipmap LOD is specified in `t.w`.

```
ret texCUBElod(s, t)
```

Parameters

Item	Description
<code>s</code>	[in] The sampler state.
<code>t</code>	[in] The texture coordinate.

Return Value

The value of the texture data.

Type Description

Name	In/Out	Template Type	Component Type	Size
<code>s</code>	in	<code>object</code>	<code>samplerCUBE</code>	1
<code>t</code>	in	<code>vector</code>	<code>float</code>	4
<code>ret</code>	out	<code>vector</code>	<code>float</code>	4

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 4	yes (pixel shader only)
Shader Model 3 (DirectX HLSL)	yes (pixel shader only)
Shader Model 2 (DirectX HLSL)	no

Shader Model	Supported
Shader Model 1 (DirectX HLSL)	no

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

texCUBEproj

Article • 08/19/2020 • 2 minutes to read

Samples a cube texture using a projective divide; the texture coordinate is divided by `t.w` before the lookup takes place.

```
ret texCUBEproj(s, t)
```

Parameters

Item	Description
<code>s</code>	[in] The sampler state.
<code>t</code>	[in] The texture coordinate.

Return Value

The value of the texture data.

Type Description

Name	In/Out	Template Type	Component Type	Size
<code>s</code>	in	object	samplerCUBE	1
<code>t</code>	in	vector	float	4
<code>ret</code>	out	vector	float	4

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 4	yes (pixel shader only)
Shader Model 3 (DirectX HLSL)	yes (pixel shader only)
Shader Model 2 (DirectX HLSL)	yes (pixel shader only)

Shader Model	Supported
Shader Model 1 (DirectX HLSL)	no

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

transpose

Article • 08/19/2020 • 2 minutes to read

Transposes the specified input matrix.

```
ret transpose(x)
```

Parameters

Item	Description
x	[in] The specified matrix.

Return Value

The transposed value of the *x* parameter.

Remarks

If the dimensions of the source matrix are *rows columns*, the resulting matrix is *columns rows*.

Type Description

Name	Template Type	Component Type	Size
x	matrix	float, int, bool	any
ret	matrix	float, int, bool	rows = same number of columns as input x, columns = same number of rows as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported

Shader Model	Supported
Shader Model 1 (DirectX HLSL) and higher shader models	yes

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

trunc

Article • 03/15/2021 • 2 minutes to read

Truncates a floating-point value to the integer component.

```
ret trunc(x)
```

Parameters

Item	Description
x	[in] The specified input.

Return Value

The input value truncated to an integer component.

Remarks

This function truncates a floating-point value to the integer component. Given a floating-point value of 1.6, the trunc function would return 1.0, whereas the [round \(DirectX HLSL\)](#) function would return 2.0.

Type Description

Name	Template Type	Component Type	Size
x	scalar , vector, or matrix	float	any
ret	Same type as input x	float	Same dimension(s) as input x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 1 (DirectX HLSL) and higher shader models	yes

Requirements

Requirement	Value
Header	Corecrt_math.h

See also

[Intrinsic Functions \(DirectX HLSL\)](#)

Asm Shader Reference

Article • 12/10/2019 • 2 minutes to read

Shaders drive the programmable graphics pipeline.

Vertex Shader Reference

- [vs_1_1](#)
- [vs_2_0](#)
- [vs_2_x](#)
- [vs_3_0](#)

[Vertex Shader Differences](#) summarizes the differences between vertex shader versions.

Pixel Shader Reference

- [ps_1_1, ps_1_2, ps_1_3, ps_1_4](#)
- [ps_2_0](#)
- [ps_2_x](#)
- [ps_3_0](#)

[Pixel Shader Differences](#) summarizes the differences between pixel shader versions.

Shader Model 4 and 5 Reference

The [Shader Model 4 Assembly](#) and [Shader Model 5 Assembly](#) sections describe the instructions that shader model 4 and 5 support.

Behavior of Constant Registers in Assembly Shaders

There are two ways to set constant registers in an assembly shader:

- Declare a shader constant in assembly code using one of the def* instructions.
- Use one of the Set***ShaderConstant* API methods.

Direct3D 9 Shader Constants

In Direct3D 9 the lifetime of defined constants in a given shader is confined to the execution of that shader only (and is non-overridable). Defined constants in Direct3D 9 have no side effects outside of the shader.

Here's an example using Direct3D 9:

Given:

Create shader1 which references c4 and defines it with the def instruction

Scenario 1:

Call Set***Shader shader1
Call Set***ShaderConstant* to set c4
Call Draw
Result: The shader will see the def'd value in c4

Given:

Scenario 1 has just completed
Create shader2 (which references c4 but does not use the def instruction to define it)

Scenario 2:

Call Set***Shader shader2
Call Draw
Result: The shader will see the value last set in c4 by Set***ShaderConstant* in scenario 1. This is because shader 2 didn't def c4.

In Direct3D 9, calling Get***ShaderConstant* will only retrieve constant values set via Set***ShaderConstant*.

Direct3D 8 Shader Constants

This behavior is different in Direct3D 8.x.

Given:

Create shader1 which references c4 and defines it with the def instruction

Scenario 1 (repeated with Direct3D 8):

Call Set***Shader with shader1
Call Set***ShaderConstant to set c4
Call Draw
Result: The shader will see the value in c4 from Set***ShaderConstant

In Direct3D 8.x `Set***ShaderConstant` takes effect immediately. Consider this scenario:

Given:

Create shader1 which references c4 and defines it with the def instruction

Scenario 3:

Call `Set***Shader` with shader1

Call Draw

Result: The shader will see the def'd value in c4

Given:

Scenario 3 has just completed

Create shader2 (which references c4 but does not use the def instruction to define it)

Scenario 4 :

Call `Set***Shader` with shader2

Call Draw

Result: The shader will see the def'd value in c4 (set by def in shader 1)

The undesirable result is that the order in which the shaders are set could affect the observed behavior of individual shaders.

Shader Driver Model Requirements

Direct3D 9 interfaces are restricted to device driver interface (DDI) drivers that are DirectX 7-level and above. To check the DDI level, run the [DirectX Diagnostic Tool](#) and examine the saved text file.

For reference, Direct3D 8 interfaces work only on DDI drivers that are DirectX 6-level and above.

Shader Binary Format

The bitwise layout of the shader instruction stream is defined in `D3d9types.h`. If you want to design your own shader compiler or construction tools and you want more information about the shader token stream, refer to the Direct3D 9 Driver Development Kit (DDK).

C-like Shader Language

See [HLSL Reference](#) to experience a C-like shader language.

Related topics

[Reference for HLSL](#)

Vertex Shaders

Article • 08/23/2019 • 2 minutes to read

- [vs_1_1](#)
- [vs_2_0](#)
- [vs_2_x](#)
- [vs_3_0](#)
- [Vertex Shader Differences](#)
- [Vertex Shader Instructions](#)
- [Vertex Shader Registers](#)

To see the newest features introduced in shader model 3, see:

- [Shader Model 3 \(DirectX HLSL\)](#)

Related topics

[Asm Shader Reference](#)

vs_1_1

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

A programmable vertex shader is made up of a set of instructions that operate on vertex data. Registers transfer data in and out of the ALU. Additional control can be applied to modify the instruction, the results, or what data gets written out.

- [Instructions - vs_1_1](#) contains a list of the available instructions.
- [Registers - vs_1_1](#) lists the different types of registers used by the vertex shader ALU.
- [Vertex Shader Register Modifiers](#) are used to modify the way an instruction works.
- [Vertex Shader Source Register Modifiers](#) alter the source register data before the instruction runs.
- [Source Register Swizzling](#) gives additional control over which register components are read, copied or written.
- [Destination Register Masking](#) determines what components of the destination register get written.

Related topics

[Vertex Shaders](#)

vs_2_0

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

A programmable vertex shader is made up of a set of instructions that operate on vertex data. Registers transfer data in and out of the ALU. Additional control can be applied to modify the instruction, the results, or what data gets written out.

- [Instructions - vs_2_0](#) contains a list of the available instructions.
- [Registers - vs_2_0](#) lists the different types of registers used by the vertex shader ALU.
- [Vertex Shader Register Modifiers](#) are used to modify the way an instruction works.
- [Vertex Shader Source Register Modifiers](#) alter the source register data before the instruction runs.
- [Source Register Swizzling](#) gives additional control over which register components are read, copied, or written.
- [Destination Register Masking](#) determines what components of the destination register get written.

Instruction Count

Each vertex shader can have up to 256 instructions stored. The number of instructions run can be much higher (because of the loop/rep support), and is capped by D3DCAPS9.MaxVShaderInstructionsExecuted, which should be at least 0xFFFF.

Related topics

[Vertex Shaders](#)

vs_2_x

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

A programmable vertex shader is made up of a set of instructions that operate on vertex data. Registers transfer data in and out of the ALU. Additional control can be applied to modify the instruction, the results, or what data gets written out.

Vertex shader version vs_2_x extends the feature set supported by vs_2_0. Each additional feature is represented by a corresponding cap in the [D3DCAPS9](#) structure within [D3DVS20CAPS](#). To use any of the enhanced features represented by these caps, the vertex shader version must be specified as vs_2_x.

- [Instructions - vs_2_x](#) contains a list of the available instructions.
- [Registers - vs_2_x](#) lists the different types of registers used by the vertex shader ALU.
- [Vertex Shader Register Modifiers](#) are used to modify the way an instruction works.
- [Vertex Shader Source Register Modifiers](#) alter the source register data before the instruction runs.
- [Source Register Swizzling](#) gives additional control over which register components are read, copied, or written.
- [Destination Register Masking](#) determines what components of the destination register get written.

New Features

New features are as follows:

Dynamic Flow Control

If [D3DVS20CAPS](#) > 0, then the following dynamic flow control instructions are supported:

- [if_comp - vs](#)
- [break - vs](#)
- [break_comp - vs](#)

If [D3DVS20CAPS](#) is also set, the following additional flow control instructions are supported:

- [setp_comp - vs](#)
- [if pred - vs](#)

- `callnz pred - vs`
- `breakp - vs`

The range of values for dynamic flow control depth is 0 to 24 and is equal to the nesting depth of the dynamic flow control instructions (see [Flow Control Nesting Limits](#) for details). If this cap is zero, the device does not support dynamic flow control instructions.

Number of Temporary Registers

`D3DVS20CAPS` represents the number of [Temporary Registers](#) supported by the device. The range of values for this cap is 12 to 32.

Static Flow Control Nesting Depth

`D3DVS20CAPS` represents the nesting depth of two types of static flow control instructions: `loop - vs/rep - vs` and `call - vs/callnz bool - vs/if bool - vs`. `loop - vs/rep - vs` instructions can be nested up to `D3DVS20CAPS` deep. Independently, `call - vs/callnz bool - vs` instructions can be nested up to `D3DVS20CAPS` deep. If `D3DVS20CAPS` is also set, then `callnz pred - vs` is counted toward the nesting depth of `call - vs/callnz bool - vs/if bool - vs` (see [Flow Control Nesting Limits](#) for details).

Predication

If `D3DVS20CAPS` is set, the device supports `setp_comp - vs` and instruction predication. If `D3DVS20CAPS` is also greater than 0, then the following additional dynamic flow control instructions are supported:

- `if pred - vs`
- `callnz pred - vs`
- `breakp - vs`

Instruction Count

Each vertex shader can have up to 256 instructions stored. The number of instructions run can be much higher (because of the loop/rep support), and is capped by `D3DCAPS9`, which should be at least 0xFFFF.

Related topics

[Vertex Shaders](#)

vs_3_0

Article • 08/19/2021 • 2 minutes to read

A programmable vertex shader is made up of a set of instructions that operate on vertex data. Registers transfer data in and out of the ALU. Additional control can be applied to modify the instruction, the results, or what data gets written out.

Vertex shader version vs_3_0 extends the feature set supported by vs_2_x. Each of the features in vs_2_X that requires a cap to be set, is available in vs_3_0 without requiring the cap.

- [Instructions - vs_3_0](#) contains a list of the available instructions.
- [Registers - vs_3_0](#) lists the different types of registers used by the vertex shader ALU.
- [Vertex Shader Register Modifiers](#) are used to modify the way an instruction works.
- [Vertex Shader Source Register Modifiers](#) alter the source register data before the instruction runs.
- [Source Register Swizzling](#) gives additional control over which register components are read, copied, or written.
- [Destination Register Masking](#) determines what components of the destination register get written.

New Features

New features of vertex shader version vs_3_0 are listed in the following sections.

Indexing Registers

In the earlier shader models, only the constant register bank could be indexed. In this model, the following register banks can be indexed, using the loop counter register (aL):

- Input register (v#)
- Output register (o#)

Vertex Textures

This shader model supports texture lookup in the vertex shader using texldl. The vertex engine has four texture sampler stages (distinct from the displacement map sampler and the texture samplers in the pixel engine) that can be used to sample textures set at those stages. See [Vertex Textures in vs_3_0 \(DirectX HLSL\)](#).

Vertex Stream Frequency

This feature allows a subset of the input registers to be initialized at a rate different from once per vertex. See [Drawing Non-Indexed Geometry](#).

Shader Output

Similar to `vs_2_0`, the output of the shader can vary with static flow control. Be careful with dynamic branching as this can cause shader outputs to vary per vertex. This will produce unpredictable results on different hardware.

Dynamic flow control

All dynamic flow control instructions are supported. The maximum nesting depth value allowed is 24. (See [Flow Control Nesting Limits](#) for details.)

Temporary Registers

A total of 32 temporary registers (`r#`) is supported.

Static Flow Control

The maximum nesting depth for `loop - vs/rep - vs` is 4. The maximum nesting depth for `call - vs/callnz bool - vs/callnz pred - vs` is 4. For `if bool - vs`, the maximum nesting depth value allowed is 24. (See [Flow Control Nesting Limits](#) for details.)

Predication

Instruction predication is supported. Use `setp_comp - vs` to set the predicate register.

Instruction Count

Each vertex shader is allowed anywhere from 512 up to the number of slots in `MaxVertexShader30InstructionSlots` in [D3DCAPS9](#). The number of instructions run can be much higher because of the loop/rep support; however, this is capped by `MaxVShaderInstructionsExecuted` in `D3DCAPS9` which should be at least `0xFFFF`.

Device Caps

If Vertex Shader 3_0 is supported, the following caps are supported in hardware (at a minimum):

Cap	Capability
Shader caps	<ul style="list-style-type: none">• DynamicFlowControlDepth is 24• NumTemps is 32• StaticFlowControlDepth is 4• Predication is supported.
GuardBandLeft, GuardBandTop, GuardBandRight, GuardBandBottom	8K
VertexShaderVersion	3_0
MaxVertexShaderConst	256
MaxVertexShader30InstructionSlots	512
Fog support	D3DPRASTERCAPS_FOGVERTEX
VertexTextureFilterCaps	<ul style="list-style-type: none">• D3DPTFILTERCAPS_MINPOINT• D3DPTFILTERCAPS_MAGPOINT
D3DDEVCAPS2_VERTEXELEMENTSCANSHARESTREAMOFFSET	Vertex elements in a vertex declaration can share the same stream offset.
Vertex formats	<ul style="list-style-type: none">• D3DDECLTYPE_UBYTE4• D3DDECLTYPE_UBYTE4N• D3DDECLTYPE_SHORT2N• D3DDECLTYPE_SHORT4N• D3DDECLTYPE_FLOAT16_2• D3DDECLTYPE_FLOAT16_4

Related topics

[Vertex Shaders](#)

Vertex Shader Differences

Article • 08/19/2020 • 2 minutes to read

Instruction Slots

Each version supports a differing number of maximum instruction slots.

Version	Maximum number of instruction slots
vs_1_1	128
vs_2_0	256
vs_2_x	256
vs_3_0	512 minimum, and up to the number of slots in D3DCAPS9.MaxVertexShader30InstructionSlots. See D3DCAPS9 .

For information about the limitations of software shaders, see [Software Shaders](#).

Flow Control Nesting Limits

- See [Flow Control Nesting Limits](#).

vs_1_1 Features

New instructions:

See [Instructions - vs_1_1](#).

New registers:

See [Registers - vs_1_1](#).

vs_2_0 Features

New features:

- Static flow control
- All four components of the [Address Register](#) (a0) are available.

New instructions:

- Setup instructions - `defb` - vs, `defi` - vs
- Arithmetic instructions - `abs` - vs, `crs` - vs, `lrp` - vs, `mova` - vs, `nrm` - vs, `pow` - vs, `sgn` - vs, `sincos` - vs
- Static flow control instructions - `call` - vs, `callnz` bool - vs, `else` - vs, `endif` - vs, `endloop` - vs, `endrep` - vs, `if` bool - vs, `label` - vs, `loop` - vs, `rep` - vs, `ret` - vs

New registers:

- Constant Boolean Register (b#)
- Constant Integer Register (i#)
- Loop Counter Register (aL)

vs_2_x Features

New features (D3DCAPS9.VS20Caps):

- Dynamic flow control
- Nesting for dynamic and static flow control instructions
- Number of `Temporary Registers` (r#) increased
- Predication

New Instructions:

- Dynamic flow control instructions - `break` - vs, `break_comp` - vs, `breakp` - vs, `callnz` pred - vs, `if_comp` - vs, `if_pred` - vs, `setp_comp` - vs

New registers:

- Predicate Register (p0)

vs_3_0 Features

New features :

- Texture lookup
- Indexable `Output Registers` (o#)
- Number of `Temporary Registers` (r#) increased to 32

New instructions:

- Setup instruction - `dcl_samplerType` (sm3 - vs asm)
- Texture instruction - `texldl` - vs

New registers:

- Sampler (Direct3D 9 asm-vs) (s#)

Related topics

[Vertex Shaders](#)

Vertex Shader Instructions

Article • 08/23/2019 • 2 minutes to read

- [Instructions - vs_1_1](#)
- [Instructions - vs_2_0](#)
- [Instructions - vs_2_x](#)
- [Instructions - vs_3_0](#)
- [Vertex Shader Register Modifiers](#)
- [Flow Control Nesting Limits](#)

Related topics

[Vertex Shaders](#)

Instructions - vs_1_1

Article • 04/19/2022 • 2 minutes to read

This section contains reference information for the vertex shader version 1_1 instructions.

There are several types of vertex shader instructions, as shown in the table. Columns to the right mean the following:

- Instruction slots - Number of instruction slots used by each instruction.
- Setup - Non-arithmetic instructions. Every shader must have a version instruction and it must be the first instruction.
- Arithmetic - These instructions provide the mathematical operations in a shader.
- New - These instructions are new to this version.

Instruction Set

Name	Description	Instruction slots	Setup	Arithmetic	New
add - vs	Add two vectors	1		x	x
dcl_usage input (sm1, sm2, sm3 - vs asm)	Declare input vertex registers (see Registers - vs_1_1)	0	x		x
def - vs	Define constants	0	x		x
dp3 - vs	Three-component dot product	1		x	x
dp4 - vs	Four-component dot product	1		x	x
dst - vs	Calculate the distance vector	1		x	x
exp - vs	Full precision 2^x	10		x	x
expf - vs	Partial precision 2^x	1		x	x
frc - vs	Fractional component	3		x	x
lit - vs	Partial lighting calculation	1		x	x
log - vs	Full precision $\log_2(x)$	10		x	x
logf - vs	Partial precision $\log_2(x)$	1		x	x

Name	Description	Instruction slots	Setup	Arithmetic	New
m3x2 - vs	3x2 multiply	2		x	x
m3x3 - vs	3x3 multiply	3		x	x
m3x4 - vs	3x4 multiply	4		x	x
m4x3 - vs	4x3 multiply	3		x	x
m4x4 - vs	4x4 multiply	4		x	x
mad - vs	Multiply and add	1		x	x
max - vs	Maximum	1		x	x
min - vs	Minimum	1		x	x
mov - vs	Move	1		x	x
mul - vs	Multiply	1		x	x
nop - vs	No operation	1		x	x
rcp - vs	Reciprocal	1		x	x
rsq - vs	Reciprocal square root	1		x	x
sge - vs	Greater than or equal compare	1		x	x
slt - vs	Less than compare	1		x	x
sub - vs	Subtract	1		x	x
vs	Version	0	x		x

Related topics

[Vertex Shader Instructions](#)

Instructions - vs_2_0

Article • 04/19/2022 • 2 minutes to read

This section contains reference information for the vertex shader version 2_0 instructions.

There are several types of vertex shader instructions, as shown in the table. Columns to the right mean the following:

- Instruction slots - Number of instruction slots used by each instruction.
- Setup - Non-arithmetic instructions. Every shader must have a version instruction and it must be the first instruction.
- Arithmetic - These instructions provide the mathematical operations in a shader.
- Flow control - These instructions add flow control capabilities such as `loop...endloop`, `if...else...endif - vs`, and subroutine calls.
- New - These instructions are new to this version.

Instruction Set

Name	Description	Instruction slots	Setup	Arithmetic	Flow control	New
abs - vs	Absolute value	1		x		x
add - vs	Add two vectors	1		x		
call - vs	Call a subroutine	2			x	x
callnz bool - vs	Call a subroutine if a Boolean register is not zero	3			x	x
crs - vs	Cross product	2		x		x
dcl_usage input (sm1, sm2, sm3 - vs asm)	Declare input vertex registers (see Registers - vs_2_0)	0		x		
def - vs	Define constants	0		x		
defb - vs	Define a Boolean constant	0		x		x
defi - vs	Define an integer constant	0		x		x

Name	Description	Instruction slots	Setup	Arithmetic	Flow	New control
dp3 - vs	Three-component dot product	1		x		
dp4 - vs	Four-component dot product	1		x		
dst - vs	Calculate the distance vector	1		x		
else - vs	Begin an <code>else - vs</code> block	1			x	x
endif - vs	End an <code>if bool - vs...else - vs</code> block	1			x	x
endloop - vs	End of a <code>loop - vs</code> block	2			x	x
endrep - vs	End of a repeat block	2			x	x
exp - vs	Full precision 2^x	1		x		
expf - vs	Partial precision 2^x	1		x		
frc - vs	Fractional component	1		x		
if bool - vs	Begin an <code>if bool - vs</code> block (using a Boolean condition)	3			x	x
label - vs	Label	0			x	x
lit - vs	Partial lighting calculation	3		x		
log - vs	Full precision $\log_2(x)$	1		x		
logf - vs	Partial precision $\log_2(x)$	1		x		
loop - vs	Loop	3			x	x
lrp - vs	Linear interpolation	2		x		x
m3x2 - vs	3x2 multiply	2		x		
m3x3 - vs	3x3 multiply	3		x		
m3x4 - vs	3x4 multiply	4		x		
m4x3 - vs	4x3 multiply	3		x		
m4x4 - vs	4x4 multiply	4		x		

Name	Description	Instruction slots	Setup	Arithmetic	Flow	New control
mad - vs	Multiply and add	1		x		
max - vs	Maximum	1		x		
min - vs	Minimum	1		x		
mov - vs	Move	1		x		
movea - vs	Move data from a floating-point register to the address register (a0)	1		x		x
mul - vs	Multiply	1		x		
nop - vs	No operation	1		x		
nrm - vs	Normalize a 4D vector	3		x		x
pow - vs	x^y	3		x		x
rcp - vs	Reciprocal	1		x		
rep - vs	Repeat	3			x	x
ret - vs	End of either a subroutine or main	1			x	x
rsq - vs	Reciprocal square root	1		x		
sge - vs	Greater than or equal compare	1		x		
sgn - vs	Sign	3		x		x
sincos - vs	Sine and cosine	8		x		x
slt - vs	Less than compare	1		x		
sub - vs	Subtract	1		x		
vs	Version	0	x			

Related topics

[Vertex Shader Instructions](#)

Instructions - vs_2_x

Article • 04/19/2022 • 3 minutes to read

This section contains reference information for the vertex shader version 2_x instructions.

There are several types of vertex shader instructions, as shown in the table. Columns to the right mean the following:

- Instruction slots - Number of instruction slots used by each instruction.
- Setup - Non-arithmetic instructions. Every shader must have a version instruction and it must be the first instruction.
- Arithmetic - These instructions provide the mathematical operations in a shader.
- Flow control - These instructions add flow control capabilities such as `loop - vs...endloop - vs`, `if bool - vs...else...endif`, and subroutine calls.
- New - These instructions are new to this version.

Instruction Set

Name	Description	Instruction slots	Setup	Arithmetic	Flow control	New
<code>abs - vs</code>	Absolute value	1		x		
<code>add - vs</code>	Add two vectors	1		x		
<code>break - vs</code>	Break out of a <code>loop - vs...endloop - vs</code> or <code>rep...endrep</code> block	1			x	x
<code>break_comp - vs</code>	Conditionally break out of a <code>loop - vs...endloop - vs</code> or <code>rep...endrep</code> block, with a comparison	3			x	x
<code>breakp - vs</code>	Break out of a <code>loop - vs...endloop - vs</code> or <code>rep...endrep</code> block, based on a predicate	3			x	x
<code>call - vs</code>	Call a subroutine	2			x	
<code>callnz bool - vs</code>	Call a subroutine if a Boolean register is not zero	3			x	

Name	Description	Instruction slots	Setup	Arithmetic	Flow	New control
callnz pred - vs	Call a subroutine if a predicate register is not zero	3			x	x
crs - vs	Cross product	2		x		
dcl_usage input (sm1, sm2, sm3 - vs)	Declare input vertex registers (see Registers - vs_2_x)	0		x		
def - vs	Define constants	0	x			
defb - vs	Define a Boolean constant	0	x			
defi - vs	Define an integer constant	0	x			
dp3 - vs	Three-component dot product	1		x		
dp4 - vs	Four-component dot product	1		x		
dst - vs	Calculate the distance vector	1		x		
else - vs	Begin an else - vs block	1			x	
endif - vs	End an if bool - vs...else - vs block	1			x	
endloop - vs	End of a loop - vs block	2			x	
endrep - vs	End of a repeat block	2			x	
exp - vs	Full precision 2^x	1		x		
expp - vs	Partial precision 2^x	1		x		
frc - vs	Fractional component	1		x		
if bool - vs	Begin an if bool - vs block (using a Boolean condition)	3			x	
if_comp - vs	Begin an if bool - vs block, with a comparison	3			x	x

Name	Description	Instruction slots	Setup	Arithmetic	Flow	New control
if pred - vs	Begin an if bool - vs block with a predicate condition	3			x	x
label - vs	Label	0			x	
lit - vs	Partial lighting calculation	3		x		
log - vs	Full precision $\log_2(x)$	1		x		
logp - vs	Partial precision $\log_2(x)$	1		x		
loop - vs	Loop	3			x	
lrp - vs	Linear interpolation	2		x		
m3x2 - vs	3x2 multiply	2		x		
m3x3 - vs	3x3 multiply	3		x		
m3x4 - vs	3x4 multiply	4		x		
m4x3 - vs	4x3 multiply	3		x		
m4x4 - vs	4x4 multiply	4		x		
mad - vs	Multiply and add	1		x		
max - vs	Maximum	1		x		
min - vs	Minimum	1		x		
mov - vs	Move	1		x		
movea - vs	Move data from a floating-point register to the address register (a0)	1		x		
mul - vs	Multiply	1		x		
nop - vs	No operation	1		x		
nrm - vs	Normalize a 4D vector	3		x		
pow - vs	x^y	3		x		
rcp - vs	Reciprocal	1		x		
rep - vs	Repeat	3			x	

Name	Description	Instruction slots	Setup	Arithmetic	Flow	New control
ret - vs	End of either a subroutine or main	1			x	
rsq - vs	Reciprocal square root	1		x		
setp_comp - vs	Set the predicate register	1			x	x
sge - vs	Greater than or equal compare	1		x		
sgn - vs	Sign	3		x		
sincos - vs	Sine and cosine	8		x		
slt - vs	Less than compare	1		x		
sub - vs	Subtract	1		x		
vs	Version	0		x		

Related topics

[Vertex Shader Instructions](#)

Instructions - vs_3_0

Article • 04/20/2022 • 3 minutes to read

This section contains reference information for the vertex shader version 3_0 instructions.

There are several types of vertex shader instructions, as shown in the table. Columns to the right mean the following:

- Instruction slots - Number of instruction slots used by each instruction.
- Setup - Non-arithmetic instructions. Every shader must have a version instruction and it must be the first instruction.
- Arithmetic - These instructions provide the mathematical operations in a shader.
- Texture - These instructions support texture address lookup.
- Flow control - These instructions add flow control such as loops, repeats, and `if bool - vs...else...endif` comparisons.
- New - These instructions are new to this version.

Instruction Set

Name	Description	Instruction slots	Setup	Arithmetic	Texture	Flow control	New control
abs - vs	Absolute value	1		x			
add - vs	Add two vectors	1		x			
break - vs	Break out of a <code>loop - vs...endloop -</code> or <code>rep...endrep</code> block	1				x	
break_comp - vs	Conditionally break out of a <code>loop - vs...endloop -</code> or <code>rep...endrep</code> block, with a comparison	3				x	

Name	Description	Instruction slots	Setup	Arithmetic	Texture	Flow	New control
breakp - vs	Break out of a loop - vs...endloop - vs or rep...endrep block, based on a predicate	3				x	
call - vs	Call a subroutine	2				x	
callnz bool - vs	Call a subroutine if a Boolean register is not zero	3				x	
callnz pred - vs	Call a subroutine if a predicate register is not zero	3				x	
crs - vs	Cross product	2			x		
dcl_usage input (sm1, sm2, sm3 - vs asm)	Declare input vertex registers (see Registers - vs_3_0)	0	x				
dcl_samplerType (sm3 - vs asm)	Declare the texture dimension for a sampler	0	x			x	
def - vs	Define constants	0	x				
defb - vs	Declare a Boolean constant	0	x				
defi - vs	Declare an integer constant	0	x				

Name	Description	Instruction slots	Setup	Arithmetic	Texture	Flow	New control
dp3 - vs	Three-component dot product	1		x			
dp4 - vs	Four-component dot product	1		x			
dst - vs	Distance	1		x			
else - vs	Begin an <code>else</code> block	1				x	
endif - vs	End an <code>if bool - vs...else</code> block	1				x	
endloop - vs	End of a <code>loop - vs</code> block	2				x	
endrep - vs	End of a repeat block	2				x	
exp - vs	Full precision 2^x	1		x			
expp - vs	Partial precision 2^x	1		x			
frc - vs	Fractional component	1		x			
if bool - vs	Begin an <code>if bool - vs</code> block (using a Boolean condition)	3				x	
if_comp - vs	Begin an <code>if bool - vs</code> block, with a comparison	3				x	
if pred - vs	Begin an <code>if bool - vs</code> block with a predicate condition	3				x	

Name	Description	Instruction slots	Setup	Arithmetic	Texture	Flow	New control
label - vs	Label	0				x	
lit - vs	Calculate lighting	3		x			
log - vs	Full precision $\log_2(x)$	1		x			
logp - vs	Partial precision $\log_2(x)$	1		x			
loop - vs	Loop	3				x	
lrp - vs	Linear interpolation	2		x			
m3x2 - vs	3x2 multiply	2		x			
m3x3 - vs	3x3 multiply	3		x			
m3x4 - vs	3x4 multiply	4		x			
m4x3 - vs	4x3 multiply	3		x			
m4x4 - vs	4x4 multiply	4		x			
mad - vs	Multiply and add	1		x			
max - vs	Maximum	1		x			
min - vs	Minimum	1		x			
mov - vs	Move	1		x			
movea - vs	Move data from a floating point register to an integer register	1		x			
mul - vs	Multiply	1		x			
nop - vs	No operation	1		x			
nrm - vs	Normalize	3		x			

Name	Description	Instruction slots	Setup	Arithmetic	Texture	Flow	New control
pow - vs	x^y	3		x			
rcp - vs	Reciprocal	1		x			
rep - vs	Repeat	3				x	
ret - vs	End of a subroutine	1				x	
rsq - vs	Reciprocal square root	1		x			
setp_comp - vs	Set the predicate register	1				x	
sge - vs	Greater than or equal compare	1		x			
sgn - vs	Sign	3		x			
sincos - vs	Sine and cosine	8		x			
slt - vs	Less than compare	1		x			
sub - vs	Subtract	1		x			
texldl - vs	Texture load with user-adjustable level-of-detail	See note 1			x		x
vs	Version	0	x				

Notes:

- if the texture is a cube map, slots = 5; otherwise slots = 2

Related topics

[Vertex Shader Instructions](#)

Flow Control Nesting Limits

Article • 08/23/2019 • 6 minutes to read

Vertex shader flow control instructions have two special restrictions. Nesting depths restrict the number of instructions that can be called inside of each other. In addition, each instruction has an instruction slot count that applies against the maximum number of instruction that a shader can support.

ⓘ Note

When you use the *_4_0_level_9_x HLSL shader profiles, you implicitly use of the **Shader Model 2.x** profiles to support Direct3D 9 capable hardware. Shader Model 2.x profiles support more limited flow control behavior than the **Shader Model 4.x** and later profiles.

Depth Count per Instruction for vs_2_0

Each instruction counts against one or more nesting depth limits. This table shows the depth count that each instruction adds or subtracts from the existing depth:

Instruction	Static nesting	Dynamic nesting	loop/rep nesting	call nesting	Static flow count
if bool - vs	0	0	0	0	1
if_comp - vs	n/a	n/a	n/a	n/a	n/a
if pred - vs	n/a	n/a	n/a	n/a	n/a
else - vs	0	0	0	0	1(if bool - vs only)
endif - vs	-1	0	0	0	0
rep - vs	0	0	1	0	1
endrep - vs	0	0	-1	0	0
loop - vs	0	0	1	0	1
endloop - vs	0	0	-1	0	0

Instruction	Static nesting	Dynamic nesting	loop/rep nesting	call nesting	Static flow count
break - vs	n/a	n/a	n/a	n/a	n/a
break_comp - vs	n/a	n/a	n/a	n/a	n/a
breakp - vs	n/a	n/a	n/a	n/a	n/a
call - vs	0	0	0	1	1
callnz bool - vs	0	0	0	1	1
callnz pred - vs	n/a	n/a	n/a	n/a	n/a
ret - vs	0	0	0	-1	0
setp_comp - vs	n/a	n/a	n/a	n/a	n/a

Nesting Depth

Nesting depth define how many instructions can be called inside of each other. Each type of instruction has one or more nesting limits:

Instruction Type	Maximum
Static nesting	Only limited by the static flow count
Dynamic nesting	n/a
loop/rep nesting	1
call nesting	1
Static flow count	16

Depth Count per Instruction for vs_2_x

Each instruction counts against one or more nesting depth limits. This table shows the depth count that each instruction adds or subtracts from the existing depth:

Instruction	Static nesting	Dynamic nesting	loop/rep nesting	call nesting	Static flow count
if bool - vs	1	0	0	0	1
if_comp - vs	0	1	0	0	0
if pred - vs	0	1	0	0	0
else - vs	0	0	0	0	1(if bool - vs only)
endif - vs	-1(if bool - vs)	-1(if pred - vs or if_comp - vs)	0	0	0
rep - vs	0	0	1	0	1
endrep - vs	0	0	-1	0	0
loop - vs	0	0	1	0	1
endloop - vs	0	0	-1	0	0
break - vs	0	0	0	0	0
break_comp - vs	0	1, -1	0	0	0
breakp - vs	0	0	0	0	0
call - vs	0	0	0	1	1
callnz bool - vs	0	0	0	1	1
callnz pred - vs	0	1	0	1	0
ret - vs	0	-1 (callnz pred - vs)	0	-1	0
setp_comp - vs	0	0	0	0	0

Nesting Depth

Nesting depth define how many instructions can be called inside of each other. Each type of instruction has one or more nesting limits:

Instruction Type	Maximum
Static nesting	Only limited by the static flow count
Dynamic nesting	0 or 24, see D3DCAPS9.VS20Caps.DynamicFlowControlDepth
loop/rep nesting	1 to 4, see D3DCAPS9.VS20Caps.StaticFlowControlDepth
call nesting	1 to 4, see D3DCAPS9.VS20Caps.StaticFlowControlDepth (independent of loop/rep limit)
Static flow count	16

Depth Count per Instruction for vs_2_sw

Each instruction counts against one or more nesting depth limits. This table shows the depth count that each instruction adds or subtracts from the existing depth:

Instruction	Static nesting	Dynamic nesting	loop/rep nesting	call nesting	Static flow count
if bool - vs	1	0	0	0	n/a
if_comp - vs	0	1	0	0	n/a
if pred - vs	0	1	0	0	n/a
else - vs	0	0	0	0	n/a
endif - vs	-1(if bool - vs)	-1(if pred - vs or if_comp - vs)	0	0	n/a
rep - vs	0	0	1	0	n/a
endrep - vs	0	0	-1	0	n/a
loop - vs	0	0	1	0	n/a
endloop - vs	0	0	-1	0	n/a
break - vs	0	0	0	0	n/a
break_comp - vs	0	1, -1	0	0	n/a

Instruction	Static nesting	Dynamic nesting	loop/rep nesting	call nesting	Static flow count
breakp - vs	0	0	0	0	n/a
call - vs	0	0	0	1	n/a
callnz bool - vs	0	0	0	1	n/a
callnz pred - vs	0	1	0	1	n/a
ret - vs	0	-1 (callnz pred - vs)	0	-1	n/a
setp_comp - vs	0	0	0	0	n/a

Nesting Depth

Nesting depth define how many instructions can be called inside of each other. Each type of instruction has one or more nesting limits:

Instruction Type	Maximum
Static nesting	24
Dynamic nesting	24
loop/rep nesting	4
call nesting	4
Static flow count	No limit

Depth Count per Instruction for vs_3_0

Each instruction counts against one or more nesting depth limits. This table shows the depth count that each instruction adds or subtracts from the existing depth:

Instruction	Static nesting	Dynamic nesting	loop/rep nesting	call nesting	Static flow count
--------------------	-----------------------	------------------------	-------------------------	---------------------	--------------------------

Instruction	Static nesting	Dynamic nesting	loop/rep nesting	call nesting	Static flow count
if bool - vs	1	0	0	0	n/a
if_comp - vs	0	1	0	0	n/a
if pred - vs	0	1	0	0	n/a
else - vs	0	0	0	0	n/a
endif - vs	-1(if bool - vs)	-1(if pred - vs or if_comp - vs)	0	0	n/a
rep - vs	0	0	1	0	n/a
endrep - vs	0	0	-1	0	n/a
loop - vs	0	0	1	0	n/a
endloop - vs	0	0	-1	0	n/a
break - vs	0	0	0	0	n/a
break_comp - vs	0	1, -1	0	0	n/a
breakp - vs	0	0	0	0	n/a
call - vs	0	0	0	1	n/a
callnz bool - vs	0	0	0	1	n/a
callnz pred - vs	0	1	0	1	n/a
ret - vs	0	-1 (callnz pred - vs)	0	-1	n/a
setp_comp - vs	0	0	0	0	n/a

Nesting Depth

Nesting depth define how many instructions can be called inside of each other. Each type of instruction has one or more nesting limits:

Instruction Type	Maximum

Instruction Type	Maximum
Static nesting	24
Dynamic nesting	24
loop/rep nesting	4
call nesting	4
Static flow count	No limit

Depth Count per Instruction for vs_3_sw

Each instruction counts against one or more nesting depth limits. This table shows the depth count that each instruction adds or subtracts from the existing depth:

Instruction	Static nesting	Dynamic nesting	loop/rep nesting	call nesting	Static flow count
if bool - vs	1	0	0	0	n/a
if_comp - vs	0	1	0	0	n/a
if pred - vs	0	1	0	0	n/a
else - vs	0	0	0	0	n/a
endif - vs	-1(if bool - vs)	-1(if pred - vs or if_comp - vs)	0	0	n/a
rep - vs	0	0	1	0	n/a
endrep - vs	0	0	-1	0	n/a
loop - vs	0	0	1	0	n/a
endloop - vs	0	0	-1	0	n/a
break - vs	0	0	0	0	n/a
break_comp - vs	0	1, -1	0	0	n/a
breakp - vs	0	0	0	0	n/a
call - vs	0	0	0	1	n/a

Instruction	Static nesting	Dynamic nesting	loop/rep nesting	call nesting	Static flow count
callnz bool - vs	0	0	0	1	n/a
callnz pred - vs	0	1	0	1	n/a
ret - vs	0	-1 (callnz pred - vs)	0	-1	n/a
setp_comp - vs	0	0	0	0	n/a

Nesting Depth

Nesting depth define how many instructions can be called inside of each other. Each type of instruction has one or more nesting limits:

Instruction Type	Maximum
Static nesting	24
Dynamic nesting	24
loop/rep nesting	4
call nesting	4
Static flow count	No limit

Related topics

[Vertex Shader Instructions](#)

Instruction modifiers (HLSL VS reference)

Article • 11/23/2019 • 2 minutes to read

Instruction modifiers affect the result of the instruction before it is written into the destination register.

_sat

Saturates (or clamps) the instruction result to [0,1] range before writing to the destination register.

For example:

```
add_sat dst, src0, src1
```

Where:

$\text{dst} = \text{clamp_between_0_and_1}(\text{src0} + \text{src1})$

The `_sat` instruction modifier costs no additional instruction slots.

If supported, the `_sat` instruction modifier can be used with any instruction except: `frc - vs`, `sincos - vs`, and `texldl - vs`.

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
<code>_sat</code>				x	x	

Related topics

[Vertex Shader Instructions](#)

abs - VS

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

Computes absolute value.

Syntax

`abs dst, src`

where

- dst is the destination register.
- src is a source register.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
abs		x	x	x	x	x

This instruction works as shown here.

```
dest.x = abs(src.x)
dest.y = abs(src.y)
dest.z = abs(src.z)
dest.w = abs(src.w)
```

Related topics

[Vertex Shader Instructions](#)

add - vs

Article • 03/09/2021 • 2 minutes to read

Adds two vectors.

Syntax

```
add dst, src0, src1
```

where

- dst is the destination register.
- src0 is a source register.
- src1 is a source register.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
add	x	x	x	x	x	x

The following code fragment shows the operations performed.

```
dest.x = src0.x + src1.x;  
dest.y = src0.y + src1.y;  
dest.z = src0.z + src1.z;  
dest.w = src0.w + src1.w;
```

Related topics

[Vertex Shader Instructions](#)

break - vs

Article • 11/20/2019 • 2 minutes to read

Break out of the current loop at the nearest [endloop - vs](#) or [endrep - vs](#).

Syntax

```
break
```

Remarks

This instruction is supported in the following versions.

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
break		x	x	x	x	x

Related topics

[Vertex Shader Instructions](#)

break_comp - vs

Article • 11/20/2019 • 2 minutes to read

Conditionally break out of the current loop at the nearest [endloop - vs](#) or [endrep - vs](#).

Syntax

```
break_comp src0, src1
```

Where:

- _comp is a comparison between the two source registers. It can be one of the following:

Syntax	Comparison
_gt	Greater than
_lt	Less than
_ge	Greater than or equal
_le	Less than or equal
_eq	Equal to
_ne	Not equal to

- src0 is a source register. Replicate swizzle is required to select a single component.
- src1 is a source register. Replicate swizzle is required to select a single component.

Remarks

This instruction is supported in the following versions.

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
break_comp		x	x	x	x	x

When the comparison is true, it breaks out of the current loop, as shown.

```
if (src0 comparison src1)
    jump to the corresponding endloop or endrep instruction;
```

Related topics

[Vertex Shader Instructions](#)

breakp - vs

Article • 11/20/2019 • 2 minutes to read

Conditionally break out of the current loop at the nearest [endloop - vs](#) or [endrep - vs](#). Use one of the components of the predicate register as a condition to determine whether or not to perform the instruction.

Syntax

```
breakp [!]p0.{x|y|z|w}
```

Where:

- [!] optional boolean NOT.
- p0 is the predicate register. See [Predicate Register](#).
- {x|y|z|w} is the required replicate swizzle on p0.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
breakp		x	x	x	x	x

Related topics

[Vertex Shader Instructions](#)

call - vs

Article • 03/09/2021 • 2 minutes to read

Performs a function call to the instruction marked with the provided label.

Syntax

```
call l#
```

where l# is a [label - vs](#) marking the beginning of the subroutine to be called.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
call		x	x	x	x	x

This instruction does the following:

1. Push address of the next instruction to the return address stack.
2. Continue execution from the instruction marked by the label.

In vertex shader 2_0, nesting calls are not allowed.

In vertex shader 2_x, the nesting depth is limited by the StaticFlowControlDepth element of the [D3DVSHADERCAPS2_0](#) structure. For more information, see [GetDeviceCaps](#).

In vertex shader 3_0, four levels of call nesting are allowed.

Only forward calls are allowed. This means that the location of the label inside the vertex shader should be after the call instruction referencing it.

If a call instruction is invoked inside [loop...endloop](#) block, the value of the [Loop Counter Register](#) (aL) is accessible inside the subroutine.

If a subroutine is referencing the [Loop Counter Register](#) (aL) located outside of the subroutine, every instance of the call to this subroutine should be surrounded by a [loop...endloop](#) block.

Related topics

[Vertex Shader Instructions](#)

callnz bool - vs

Article • 03/09/2021 • 2 minutes to read

Call if not zero. Performs a conditional call to the instruction marked by the label index.

Syntax

```
callnz l#, [!]b#
```

where:

- where l# is a [label - vs](#) marking the beginning of the subroutine to be called.
- [!] is the optional boolean NOT modifier.
- b# identifies a [Constant Boolean Register](#).

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
callnz bool		x	x	x	x	x

This instruction does the following:

```
if (specified boolean register is not zero)
{
    Push address of the next instruction to the return address stack.
    Continue execution from the instruction marked by the label.
}
```

This instruction consumes one vertex shader instruction slot.

Related topics

[Vertex Shader Instructions](#)

callnz pred - vs

Article • 11/20/2019 • 2 minutes to read

Call if not zero, with a predicate. Performs a conditional call to the instruction marked by the label index. Predication uses a boolean value to determine whether or not to perform the instruction.

Syntax

```
callnz l#, [!]p0.{x|y|z|w}
```

where:

- l# is a [label - vs](#) marking the beginning of the subroutine to be called.
- [!] is an optional negate modifier.
- p0 is the [Predicate Register](#).
- {x|y|z|w} is the required replicate swizzle on p0.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
callnz pred		x	x	x	x	x

This instruction does the following:

```
if (specified register component is not zero)
{
    Push address of the next instruction to the return address stack.
    Continue execution from the instruction marked by the label.
}
```

This instruction consumes one vertex shader instruction slot.

Related topics

Vertex Shader Instructions

CRS - VS

Article • 03/09/2021 • 2 minutes to read

Computes a cross product using the right-hand rule.

Syntax

```
crs dst, src0, src1
```

where

- dst is the destination register.
- src0 is a source register.
- src1 is a source register.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
crs		x	x	x	x	x

This instruction works as shown here.

```
dest.x = src0.y * src1.z - src0.z * src1.y;  
dest.y = src0.z * src1.x - src0.x * src1.z;  
dest.z = src0.x * src1.y - src0.y * src1.x;
```

Some restrictions on use:

- src0 cannot be the same register as dest.
- src1 cannot be the same register as dest.
- src0 cannot have any swizzle other than the default swizzle (.xyzw).
- src1 cannot have any swizzle other than the default swizzle (.xyzw).
- dest has to have exactly one of the following seven masks: .x | .y | .z | .xy | .xz | .yz | .xyz.
- dest must be a temporary register.

- dest must not be the same register as src0 or src1

Related topics

[Vertex Shader Instructions](#)

dcl_samplerType (sm3 - vs asm)

Article • 06/30/2021 • 2 minutes to read

Declare a vertex shader sampler.

Syntax

```
dcl_samplerType s#
```

where:

- `_samplerType` defines the sampler data type. This determines how many coordinates are required by each texture coordinate when sampling. The following texture coordinate dimensions are defined.
 - `_2d`
 - `_cube`
 - `_volume`
- `s#` identifies a sampler where `s` is an abbreviation for the sampler, and `#` is the sampler number. [Sampler \(Direct3D 9 asm-vs\)](#)s are pseudo registers because you cannot directly read or write to them.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
dcl_samplerType				x	x	

All `dcl_samplerType` instructions must appear before the first executable instruction.

Related topics

[Vertex Shader Instructions](#)

dcl_usage input (sm1, sm2, sm3 - vs asm)

Article • 06/30/2021 • 2 minutes to read

Declare the association between a vertex element usage and a usage index for a vertex shader input register.

Syntax

```
dcl_usage[usage_index] v#
```

Where:

- dcl_usage identifies how the register data will be used. This is the same value as the members of [D3DDECLUSAGE](#) without the D3DDECLUSAGE prefix.
- usage_index is an optional integer index between 0 and 15. It modifies the usage data. The index matches the usage index in a vertex declaration. See [Vertex Declaration \(Direct3D 9\)](#). The index is appended to the usage value (dcl_usage) with no space. If it is not supplied, it is assumed to be 0.
- v# is a [Input Register](#).

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
dcl_usage	x	x	x	x	x	x

All dcl_usage instructions must appear before the first executable instruction.

Related topics

[Vertex Shader Instructions](#)

dcl_usage output (sm1, sm2, sm3 - vs asm)

Article • 04/26/2022 • 2 minutes to read

The various types of output registers have been collapsed into twelve output registers (two for color, eight for texture, one for position, and one for fog and point size). These can be used for anything the user wants to interpolate for the pixel shader: texture coordinates, colors, fog, and so on.

Output registers require declarations that include semantics. For instance, the old position and point size registers are replaced by declaring an output register with a position or point-size semantic.

Of the twelve output registers, any ten (not necessarily o0 to o9) have four components (xyzw), another one must be declared as position (and must also include all four components), and optionally one more can be a scalar point size.

Syntax

The syntax for declaring output registers is similar to the declarations for the input register:

- dcl_semantics o[.write_mask]

Where:

- dcl_semantics can use the same set of semantics as for the input declaration. Semantic names come from [D3DDECLUSAGE](#) (and are paired with an index, such as position3). There always has to be one output register with the position0 semantic when not used for processing vertices. The position0 semantic and the pointsize0 semantic are the only ones that have meaning beyond simply allowing linkage from vertex to pixel shaders. For shaders with flow control, it is assumed that the worst case output is declared. There are no defaults if a shader does not actually output what it declares it should (due to flow control).
- o is an output register. See [Output_Registers](#).
- write_mask indicates the same output register that can be declared multiple times (so different semantics can be applied to individual components), each time with a unique write mask. However, the same semantic cannot be used multiple times in a declaration. This means that vectors must be four components or less, and cannot go across four-component register boundaries (individual registers). When the

point-size semantic is used, it should have full write mask because it is considered a scalar. When the position semantic is used, it should have a full write mask because all four components have to be written.

Remarks

Vertex shader versions	3_0	3_sw
dcl_usage	x	x

All `dcl_usage` instructions must appear before the first executable instruction.

Declaration Examples

```
vs_3_0
dcl_color4    o3.x      // color4 is a semantic name.
dcl_texcoord3 o3.yz     // Different semantics can be packed into one
register.
dcl_fog        o3.w
dcl_tangent    o4.xyz
dcl_position   o7.xyzw // position must be declared to some unique register
                      // in a vertex shader, with all 4 components.

dcl_psize      o6       // Pointsize cannot have a mask
                      // (that is, mask is full .xyzw)
                      // This is an implied scalar register.
                      // No other semantics can be assigned to any
components
                      // of this register.
                      // If pointsize declaration is not used (typical),
                      // only 11 "out" registers are available, not 12.
                      // Pixel shaders cannot see this value.
```

Related topics

[Vertex Shader Instructions](#)

def - vs

Article • 08/19/2020 • 2 minutes to read

Defines vertex shader constants.

Syntax

```
def dst, float1, float2, float3, float4
```

where

- dst is the destination register.
- float1, float2, float3, float4 are four floating point numbers.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
def	x	x	x	x	x	x

The def instruction defines a shader constant whose value is loaded anytime a shader is set to a device. These are called immediate constants. Immediate constants take precedence over constants set by API methods `SetVertexShaderConstantF`.

There are two ways to set a constant in a shader.

1. Use def - vs to define the constant directly inside a shader.

def - vs can only define floating-point constants.

2. Use the API methods to set a constant.

- Use [SetVertexShaderConstantF](#) to set a floating-point constant.

Related topics

[Vertex Shader Instructions](#)

defi - vs

defb - vs

defb - vs

Article • 08/19/2020 • 2 minutes to read

Defines a Boolean constant value, which should be loaded anytime a shader is set to a device.

Syntax

```
defb dest, booleanValue
```

where

- dst is the destination register.
- booleanValue is a boolean value, either True or False.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
defb		x	x	x	x	x

The defb - vs instruction defines a boolean shader constant whose value is loaded anytime a shader is set to a device. These are called immediate constants. Immediate constants take precedence over constants set by the API method [SetVertexShaderConstantB](#).

There are two ways to set a boolean constant in a shader.

1. Use defb - vs to define the constant directly inside a shader.
2. Use the API methods to set a constant.
 - Use [SetVertexShaderConstantB](#) to set a Boolean constant.

Related topics

[Vertex Shader Instructions](#)

def - vs

defi - vs

defi - vs

Article • 08/19/2020 • 2 minutes to read

Defines an integer constant value, which should be loaded anytime a shader is set to a device.

Syntax

```
defi dst, integerValue0, integerValue1, integerValue2, integerValue3
```

- dst is the destination register.
- integerValue# is a constant integer value.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
defi		x	x	x	x	x

The defi instruction defines an integer shader constant whose value is loaded anytime a shader is set to a device. These are called immediate constants. Immediate constants take precedence over constants set by the API method [SetVertexShaderConstant1](#).

There are two ways to set an integer constant in a shader.

1. Use defi to define the integer constant vector directly inside a shader.
2. Use the API methods to set a constant.

- Use [SetVertexShaderConstant1](#) to set an integer constant.

Related topics

[Vertex Shader Instructions](#)

[def - vs](#)

[defb - vs](#)

dp3 - vs

Article • 03/09/2021 • 2 minutes to read

Computes the three-component dot product of the source registers.

Syntax

```
dp3 dst, src0, src1
```

where

- dst is the destination register.
- src0 is a source register.
- src1 is a source register.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
dp3	x	x	x	x	x	x

The following code fragment shows the operations performed:

```
dest.w = (src0.x * src1.x) + (src0.y * src1.y) + (src0.z * src1.z);  
dest.x = dest.y = dest.z = dest.w;
```

Related topics

[Vertex Shader Instructions](#)

dp4 - vs

Article • 03/09/2021 • 2 minutes to read

Computes the four-component dot product of the source registers.

Syntax

```
dp4 dst, src0, src1
```

where

- dst is the destination register.
- src0 is a source register.
- src1 is a source register.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
dp4	x	x	x	x	x	x

The following code fragment shows the operations performed:

```
dest.w = (src0.x * src1.x) + (src0.y * src1.y) +
          (src0.z * src1.z) + (src0.w * src1.w);
dest.x = dest.y = dest.z = dest.w;
```

Related topics

[Vertex Shader Instructions](#)

dst - vs

Article • 03/09/2021 • 2 minutes to read

Calculates a distance vector.

Syntax

```
dst dest, src0, src1
```

where

- dest is the destination register.
- src0 is a source register.
- src1 is a source register.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
dst	x	x	x	x	x	x

The following code fragment shows the operations performed:

```
dest.x = 1;
dest.y = src0.y * src1.y;
dest.z = src0.z;
dest.w = src1.w;
```

The first source operand (src0) is assumed to be the vector (ignored, $d \cdot d$, $d \cdot d$, ignored) and the second source operand (src1) is assumed to be the vector (ignored, $1/d$, ignored, $1/d$). The destination (dest) is the result vector ($1, d, d \cdot d, 1/d$).

Related topics

[Vertex Shader Instructions](#)

else - vs

Article • 03/09/2021 • 2 minutes to read

Start of an else block.

Syntax

```
else
```

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
else		x	x	x	x	x

If the condition in the corresponding [if bool - vs](#) statement is true, the code enclosed by the if bool - vs statement and the matching [endif](#) is run. Otherwise, the code enclosed by the else...endif statements is run.

Related topics

[Vertex Shader Instructions](#)

[if bool - vs](#)

[endif - vs](#)

endif - vs

Article • 11/20/2019 • 2 minutes to read

Marks the end of an [if bool - vs...else](#) block.

Syntax

```
endif
```

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
endif		x	x	x	x	x

Related topics

[Vertex Shader Instructions](#)

[if bool - vs](#)

[else - vs](#)

endloop - vs

Article • 11/20/2019 • 2 minutes to read

End of a [loop...endloop](#) block.

Syntax

```
endloop
```

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
endloop		x	x	x	x	x

This instruction works as shown here.

```
LoopCounter += LoopStep;
LoopIterationCount = LoopIterationCount - 1;
if (LoopIterationCount > 0)
    Continue execution at the StartLoopOffset
```

endloop must follow the last instruction of a [loop - vs](#) block.

Example

```
loop aL, i3
    add r1, r0, c2[ aL ]
endloop
```

Related topics

Vertex Shader Instructions

endrep - vs

Article • 11/20/2019 • 2 minutes to read

End a [rep - vs](#)...endrep block.

Syntax

```
endrep
```

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
endrep		x	x	x	x	x

Example

```
rep i2
    add r0, r0, c0
endrep
```

Related topics

[Vertex Shader Instructions](#)

exp - VS

Article • 03/09/2021 • 2 minutes to read

Provides full precision exponential 2^x .

Syntax

```
exp dst, src
```

where

- dst is the destination register.
- src is a source register. Source register requires explicit use of replicate swizzle, that is, exactly one of the .x, .y, .z, .w swizzle components (or the .r, .g, .b, .a equivalents) must be specified. See [Source Register Swizzling](#).

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
exp	x	x	x	x	x	x

This instruction provides at least 21 bits of precision.

The following code fragment shows the operations performed:

```
dest.x = dest.y = dest.z = dest.w = (float)pow(2,  
src.replicateSwizzleComponent);
```

Related topics

[Vertex Shader Instructions](#)

expp - VS

Article • 11/20/2019 • 2 minutes to read

Provides partial precision exponential 2^x .

Syntax

```
expp dst, src.{x|y|z|w}
```

Where:

- dst is the destination register.
- src is a source register. Source register requires explicit use of replicate swizzle, that is, exactly one of the .x, .y, .z, .w swizzle components (or the .r, .g, .b, .a equivalents) must be specified.
- {x|y|z|w} is the required replicate swizzle on source register.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
expp	x	x	x	x	x	x

vs_1_1

The [exp - vs](#) instruction operates differently depending on vertex shader versions.

In vs_1_1, the expp instruction gives the following results:

```
v = the scalar value from the source register with a replicate swizzle

dest.x = pow(2, floor(v))
dest.y = v - floor(v)
dest.z = pow(2, v) (partial-precision)
dest.w = 1
```

In vs_2_0 and up, the expp instruction gives the following results:

```
v = the scalar value from the source register with a replicate swizzle  
dest.x = dest.y = dest.z = dest.w = pow(2, v) (partial-precision)
```

vs_2_0

In vs_2_0 and up, the instruction works like this:

```
float V = the scalar value from the source register with a replicate swizzle  
dest.x = dest.y = dest.z = dest.w = pow( 2, V ) (partial-precision)
```

The instruction provides at least 10 bits of precision.

Related topics

[Vertex Shader Instructions](#)

frc - VS

Article • 03/09/2021 • 2 minutes to read

Returns the fractional portion of each input component.

Syntax

```
frc dst, src
```

where

- dst is the destination register.
- src is a source register.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
frc	x	x	x	x	x	x

The following code fragment shows conceptually how the instruction operates.

```
dest.x = src.x - (float)floor(src.x);
dest.y = src.y - (float)floor(src.y);
dest.z = src.z - (float)floor(src.z);
dest.w = src.w - (float)floor(src.w);
```

The floor function converts the argument passed in to the greatest integer that is less than (or equal to) the argument. This is converted to a float and then subtracted from the original value. The resulting fractional value ranges from 0.0 through 1.0.

For version 1_1, the allowable write masks are .y and .xy (.x is not allowed).

Related topics

Vertex Shader Instructions

if bool - vs

Article • 11/20/2019 • 2 minutes to read

Starts an if...[else...endif - vs](#) block.

Syntax

```
if bool
```

where bool is a bool register number. See [Constant Boolean Register](#).

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
if bool	x	x	x	x	x	x

If the source Boolean register in the if statement is true, the code enclosed by the if statement and the matching else is run. Otherwise, the code enclosed by the [else...endif - vs](#) statements is run. This instruction consumes one instruction slot.

if blocks can be nested.

An if block cannot straddle a loop block.

Example

This instruction provides conditional static flow control.

```
defb b2, TRUE  
...  
if b2  
// Instructions to run if b2 is nonzero  
else
```

```
// Instructions to run otherwise  
endif
```

Related topics

[Vertex Shader Instructions](#)

[else - vs](#)

[endif - vs](#)

if_comp - vs

Article • 11/20/2019 • 2 minutes to read

Start an `if bool - vs...else - vs...endif - vs` block, with a condition based on values that could be computed in a shader. This instruction is used to skip a block of code, based on a condition.

Syntax

```
if_comp src0, src1
```

Where:

- `_comp` is a comparison between the two source registers. It can be one of the following:

Syntax	Comparison
<code>_gt</code>	Greater than
<code>_lt</code>	Less than
<code>_ge</code>	Greater than or equal
<code>_le</code>	Less than or equal
<code>_eq</code>	Equal to
<code>_ne</code>	Not equal to

- `src0` is a source register. Replicate swizzle is required to select a component.
- `src1` is a source register. Replicate swizzle is required to select a component.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
<code>if_comp</code>		x	x	x	x	x

This instruction is used to skip a block of code, based on a condition.

```
if_lt src0, src1  
    jump to the corresponding else or endif instruction;
```

Be careful using the equals and not equals comparison modes on floating point numbers. Because rounding occurs during floating point calculations, the comparison can be done against an epsilon value (small nonzero number) to avoid errors.

Restrictions include:

- if_comp...[else - vs](#)...[endif - vs](#) blocks (along with the predicated if blocks) can be nested up to 24 layers deep.
- src0 and src1 registers require a replicate swizzle.
- if_comp blocks must end with an [else - vs](#) or [endif - vs](#) instruction.
- if_comp...[else - vs](#)...[endif - vs](#) blocks cannot straddle a loop block. The if_comp block must be completely inside, or outside the [loop - vs](#) block.

Example

This instruction provides conditional dynamic flow control.

```
if_lt r3.x, r4.y  
    // Instructions to run if r3.x < r4.y  
  
else  
    // Instructions to run otherwise  
  
endif
```

Related topics

[Vertex Shader Instructions](#)

if pred - vs

Article • 11/20/2019 • 2 minutes to read

Start of an if pred - vs...[else - vs](#)...[endif - vs](#) block, with the condition taken from the content of the predicate register.

Syntax

```
if [!]pred.replicateSwizzle
```

Where:

- [!] an optional NOT modifier. This modifies the value in the predicate register.
- pred is the predicate register, p0. See [Predicate Register](#).
- replicateSwizzle is a single component that is copied (or replicated) to all four components (swizzled). Valid components are: x, y, z, w or r, g, b, a.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
if pred		x	x	x	x	x

This instruction is used to skip a block of code, based on a channel of the predicate register. Each if_pred block must end with an else or endif instruction.

Restrictions include:

if_pred blocks can be nested. This counts to the total dynamic nesting depth along with [if_comp](#) blocks.

An if_pred block cannot straddle a loop block, it should be either completely inside it or surround it.

Related topics

[Vertex Shader Instructions](#)

label - vs

Article • 03/09/2021 • 2 minutes to read

Mark the next instruction as having a label index.

Syntax

label l#

where # identifies the label number.

For `vs_2_0`, and `vs_2_x`, the label number must be between between 0 and 15.

For `vs_2_sw`, `vs_3_0` and `vs_3_sw`, the label number must be between between 0 and 2047.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
label		x	x	x	x	x

This instruction defines a label located at the next shader instruction. The label instruction can only be present directly after a [ret](#) instruction (which defines the end of the previous subroutine or main program).

Related topics

[Vertex Shader Instructions](#)

lit - vs

Article • 11/20/2019 • 2 minutes to read

Provides partial support for lighting by calculating lighting coefficients from two dot products and an exponent.

Syntax

```
lit dst, src
```

where

- dst is the destination register.
- src is a source register.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
lit	x	x	x	x	x	x

The source vector is assumed to contain the values shown in the following pseudocode.

```
src.x = N*L      ; The dot product between normal and direction to light  
src.y = N*H      ; The dot product between normal and half vector  
src.z = ignored  ; This value is ignored  
src.w = exponent ; The value must be between -128.0 and 128.0
```

The following code fragment shows the operations performed.

```
dest.x = 1;  
dest.y = 0;  
dest.z = 0;  
dest.w = 1;
```

```
float power = src.w;
const float MAXPOWER = 127.9961f;
if (power < -MAXPOWER)
    power = -MAXPOWER;           // Fits into 8.8 fixed point format
else if (power > MAXPOWER)
    power = MAXPOWER;          // Fits into 8.8 fixed point format

if (src.x > 0)
{
    dest.y = src.x;
    if (src.y > 0)
    {
        // Allowed approximation is EXP(power * LOG(src.y))
        dest.z = (float)(pow(src.y, power));
    }
}
```

Reduced precision arithmetic is acceptable in evaluating the destination y component (dest.y). An implementation must support at least eight fraction bits in the power argument. Dot products are calculated with normalized vectors, and clamp limits are -128 to 128.

Error should correspond to a [logp - vs](#) and [exp - vs](#) combination, or no more than approximately one significant bit for an 8-bit color component.

Related topics

[Vertex Shader Instructions](#)

log - VS

Article • 03/09/2021 • 2 minutes to read

Full precision $\log_2(x)$.

Syntax

```
log dst, src
```

where

- dst is the destination register.
- src is a source register. Source register requires explicit use of replicate swizzle, that is, exactly one of the .x, .y, .z, .w swizzle components (or the .r, .g, .b, .a equivalents) must be specified.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
log	x	x	x	x	x	x

The following code fragment shows the operations performed.

```
float v = abs(src);
if (v != 0)
{
    dest.x = dest.y = dest.z = dest.w =
        (float)(log(v)/log(2));
}
else
{
    dest.x = dest.y = dest.z = dest.w = -FLT_MAX;
}
```

This instruction accepts a scalar source whose sign bit is ignored. The result is replicated to all four channels.

This instruction provides 21 bits of precision.

Related topics

[Vertex Shader Instructions](#)

logp - vs

Article • 11/20/2019 • 2 minutes to read

Partial precision $\log_2(x)$.

Syntax

```
logp dst, src
```

where

- dst is the destination register.
- src is a source register. Source register requires explicit use of replicate swizzle, that is, exactly one of the .x, .y, .z, .w swizzle components (or the .r, .g, .b, .a equivalents) must be specified.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
logp	x	x	x	x	x	x

The following code fragment shows the operations performed.

```
float f = abs(src);
if (f != 0)
    dest.x = dest.y = dest.z = dest.w = (float)(log(f)/log(2));
else
    dest.x = dest.y = dest.z = dest.w = -FLT_MAX;
```

This instruction provides logarithm base 2 partial precision, up to 10 bits.

Related topics

[Vertex Shader Instructions](#)

loop - VS

Article • 11/20/2019 • 2 minutes to read

Start a loop...[endloop](#) block.

Syntax

```
loop aL, i#
```

Where:

- aL is the [Loop Counter Register](#) holding the current loop count.
- i# is a [Constant Integer Register](#). See remarks.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
loop		x	x	x	x	x

- The [Loop Counter Register](#) (aL) holds the current loop count, and can be used for relative addressing into [Constant Integer Register](#) (c#) or [Output Registers](#) (o#) inside the loop block.
- i#.x specifies the iteration count. The legal range is [0, 255]. Note that this instruction does not increment or decrement the value of i#.x.
- i#.y specifies the initial value of the [Loop Counter Register](#) (aL) register. The legal range is [0, 255]. Note that this instruction does not increment or decrement the value of i#.y.
- i#.z specifies the step/stride size. The legal range is [-128, 127].
- i#.w is not used and must be set to 0.
- Loop blocks may be nested. See [Flow Control Nesting Limits](#).
- When nested, the value of the [Loop Counter Register](#) (aL) refers to the immediate enclosing loop block.
- Loop blocks are allowed to be either completely inside an if* block or completely surrounding it. No straddling is allowed.

Example

```
loop aL, i3
    add r1, r0, c2[aL]
endloop
```

Related topics

[Vertex Shader Instructions](#)

lrp - vs

Article • 03/09/2021 • 2 minutes to read

Interpolates linearly between the second and third source registers by a proportion specified in the first source register.

Syntax

```
lrp dst, src0, src1, src2
```

where

- dst is the destination register.
- src0 is a source register.
- src1 is a source register.
- src2 is a source register.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
lrp		x	x	x	x	x

This instruction performs the linear interpolation based on the following formula.

```
dest.x = src0.x * (src1.x - src2.x) + src2.x;  
dest.y = src0.y * (src1.y - src2.y) + src2.y;  
dest.z = src0.z * (src1.z - src2.z) + src2.z;  
dest.w = src0.w * (src1.w - src2.w) + src2.w;
```

Related topics

[Vertex Shader Instructions](#)

m3x2 - vs

Article • 03/09/2021 • 2 minutes to read

Multiplies a 3-component vector by a 3x2 matrix.

Syntax

```
m3x2 dst, src0, src1
```

where

- dst is the destination register. Result is a 2-component vector.
- src0 is a source register representing a 3-component vector.
- src1 is a source register representing a 3x2 matrix, which corresponds to the first of 2 consecutive registers.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
m3x2	x	x	x	x	x	x

The xy mask is required for the destination register. Negate and swizzle modifiers are allowed for src0 but not for src1.

The following equations show how the instruction operates:

```
dest.x = (src0.x * src1.x) + (src0.y * src1.y) + (src0.z * src1.z);
dest.y = (src0.x * src2.x) + (src0.y * src2.y) + (src0.z * src2.z);
```

The input vector is in register src0. The input 3x2 matrix is in register src1 and the next higher register in the same register file, as shown in the following expansion. A 2D result is produced, leaving the other elements of the destination register (dest.z and dest.w) unaffected.

This operation is commonly used for 2D transforms. This instruction is implemented as a pair of dot products as shown here.

```
m3x2    r0.xy, r1, c0    which will be expanded to:
```

```
dp3    r0.x, r1, c0  
dp3    r0.y, r1, c1
```

Swizzle and negate modifiers are invalid for the src1 register. The dest and src0 register, or any of src1+i registers, cannot be the same.

Related topics

[Vertex Shader Instructions](#)

m3x3 - vs

Article • 03/09/2021 • 2 minutes to read

Multiplies a 3-component vector by a 3x3 matrix.

Syntax

```
m3x3 dst, src0, src1
```

where

- dst is the destination register. Result is a 3-component vector.
- src0 is a source register representing a 3-component vector.
- src1 is a source register representing a 3x3 matrix, which corresponds to the first of 3 consecutive registers.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
m3x3	x	x	x	x	x	x

The xyz mask is required for the destination register. Negate and swizzle modifiers are allowed for src0 but not for src1.

The following code fragment shows the operations performed.

```
dest.x = (src0.x * src1.x) + (src0.y * src1.y) + (src0.z * src1.z);
dest.y = (src0.x * src2.x) + (src0.y * src2.y) + (src0.z * src2.z);
dest.z = (src0.x * src3.x) + (src0.y * src3.y) + (src0.z * src3.z);
```

The input vector is in register src0. The input 3x3 matrix is in register src1, and the next two higher registers, as shown in the expansion below. A 3D result is produced, leaving the other element of the destination register (dest.w) unaffected.

This operation is commonly used for transforming normal vectors during lighting computations. This instruction is implemented as a pair of dot products as shown below.

```
m3x3 r0.xyz, r1, c0 which will be expanded to:
```

```
dp3    r0.x, r1, c0  
dp3    r0.y, r1, c1  
dp3    r0.z, r1, c2
```

Related topics

[Vertex Shader Instructions](#)

m3x4 - vs

Article • 03/09/2021 • 2 minutes to read

Multiplies a 3-component vector by a 3x4 matrix.

Syntax

```
m3x4 dst, src0, src1
```

where

- dst is the destination register. Result is a 4-component vector.
- src0 is a source register representing a 3-component vector.
- src1 is a source register representing a 3x4 matrix, which corresponds to the first of 4 consecutive registers.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
m3x4	x	x	x	x	x	x

The xyzw (default) mask is required for the destination register. Negate and swizzle modifiers are allowed for src0 but not for src1.

The following code fragment shows the operations performed.

```
dest.x = (src0.x * src1.x) + (src0.y * src1.y) + (src0.z * src1.z);
dest.y = (src0.x * src2.x) + (src0.y * src2.y) + (src0.z * src2.z);
dest.z = (src0.x * src3.x) + (src0.y * src3.y) + (src0.z * src3.z);
dest.w = (src0.x * src4.x) + (src0.y * src4.y) + (src0.z * src4.z);
```

The input vector is in register src0. The input 3x4 matrix is in register src1 and the next three higher registers, as shown in the expansion below.

This operation is commonly used for transforming a position vector by a matrix that has a projective effect but applies no translation. This instruction is implemented as a pair of dot products as shown here.

```
m3x4    r0.xyzw, r1, c0    will be expanded to:
```

```
dp3 r0.x, r1, c0  
dp3 r0.y, r1, c1  
dp3 r0.z, r1, c2  
dp3 r0.w, r1, c3
```

Related topics

[Vertex Shader Instructions](#)

m4x3 - VS

Article • 03/09/2021 • 2 minutes to read

Multiplies a 4-component vector by a 4x3 matrix.

Syntax

```
m4x3 dst, src0, src1
```

where

- dst is the destination register. Result is a 3-component vector.
- src0 is a source register representing a 4-component vector.
- src1 is a source register representing a 4x3 matrix, which corresponds to the first of 3 consecutive registers.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
m4x3	x	x	x	x	x	x

The xyz mask is required for the destination register. Negate and swizzle modifiers are allowed for src0, but not for src1.

The following code fragment shows the operations performed.

```
dest.x = (src0.x * src1.x) + (src0.y * src1.y) + (src0.z * src1.z) + (src0.w * src1.w);
dest.y = (src0.x * src2.x) + (src0.y * src2.y) + (src0.z * src2.z) + (src0.w * src2.w);
dest.z = (src0.x * src3.x) + (src0.y * src3.y) + (src0.z * src3.z) + (src0.w * src3.w);
```

The input vector is in register src0. The input 4x3 matrix is in register src1, and the next two higher registers, as shown in the expansion below. A 3D result is produced, leaving

the other element of the destination register (dest.w) unaffected.

This operation is commonly used for transforming a position vector by a matrix that has no projective effect, such as occurs in model-space transformations. This instruction is implemented as a pair of dot products as shown below.

```
m4x3    r0.xyz, r1, c0    will be expanded to:
```

```
dp4    r0.x, r1, c0  
dp4    r0.y, r1, c1  
dp4    r0.z, r1, c2
```

Swizzle and negate modifiers are invalid for the src1 register. The dst and src0 register cannot be the same.

Related topics

[Vertex Shader Instructions](#)

m4x4 - vs

Article • 03/09/2021 • 2 minutes to read

Multiplies a 4-component vector by a 4x4 matrix.

Syntax

```
m4x4 dst, src0, src1
```

where

- dst is the destination register. Result is a 4-component vector.
- src0 is a source register representing a 4-component vector.
- src1 is a source register representing a 4x4 matrix, which corresponds to the first of 4 consecutive registers.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
m4x4	x	x	x	x	x	x

The xyzw (default) mask is required for the destination register. Negate and swizzle modifiers are allowed for src0, but not for src1.

Swizzle and negate modifiers are invalid for the src0 register. The dest and src0 registers cannot be the same.

The following code fragment shows the operations performed.

```
dest.x = (src0.x * src1.x) + (src0.y * src1.y) + (src0.z * src1.z) +
         (src0.w * src1.w);
dest.y = (src0.x * src2.x) + (src0.y * src2.y) + (src0.z * src2.z) +
         (src0.w * src2.w);
dest.z = (src0.x * src3.x) + (src0.y * src3.y) + (src0.z * src3.z) +
         (src0.w * src3.w);
```

```
dest.w = (src0.x * src4.x) + (src0.y * src4.y) + (src0.z * src4.z) +  
        (src0.w * src4.w);
```

The input vector is in register src0. The input 4x4 matrix is in register src1, and the next three higher registers, as shown in the expansion below.

This operation is commonly used for transforming a position by a projection matrix. This instruction is implemented as a series of dot products as shown here.

```
m4x4    r0.xyzw, r1, c0    will be expanded to:
```

```
dp4    r0.x, r1, c0  
dp4    r0.y, r1, c1  
dp4    r0.z, r1, c2  
dp4    r0.w, r1, c3
```

Related topics

[Vertex Shader Instructions](#)

mad - vs

Article • 11/20/2019 • 2 minutes to read

Multiplies and adds sources.

Syntax

```
mad dst, src0, src1, src2
```

where

- dst is the destination register.
- src0 is a source register.
- src1 is a source register.
- src2 is a source register.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
mad	x	x	x	x	x	x

The following code fragment shows the operations performed.

```
dest.x = src0.x * src1.x + src2.x;  
dest.y = src0.y * src1.y + src2.y;  
dest.z = src0.z * src1.z + src2.z;  
dest.w = src0.w * src1.w + src2.w;
```

Related topics

[Vertex Shader Instructions](#)

max - VS

Article • 03/09/2021 • 2 minutes to read

Calculates the maximum of the sources.

Syntax

```
max dst, src0, src1
```

where

- dst is the destination register.
- src0 is a source register.
- src1 is a source register.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
max	x	x	x	x	x	x

The following code fragment shows the operations performed.

```
dest.x=(src0.x >= src1.x) ? src0.x : src1.x;  
dest.y=(src0.y >= src1.y) ? src0.y : src1.y;  
dest.z=(src0.z >= src1.z) ? src0.z : src1.z;  
dest.w=(src0.w >= src1.w) ? src0.w : src1.w;
```

Related topics

[Vertex Shader Instructions](#)

min - VS

Article • 03/09/2021 • 2 minutes to read

Calculates the minimum of the sources.

Syntax

```
min dst, src0, src1
```

where

- dst is the destination register.
- src0 is a source register.
- src1 is a source register.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
min	x	x	x	x	x	x

The following code fragment shows the operations performed.

```
dest.x=(src0.x < src1.x) ? src0.x : src1.x;
dest.y=(src0.y < src1.y) ? src0.y : src1.y;
dest.z=(src0.z < src1.z) ? src0.z : src1.z;
dest.w=(src0.w < src1.w) ? src0.w : src1.w;
```

Related topics

[Vertex Shader Instructions](#)

MOV - VS

Article • 11/20/2019 • 2 minutes to read

Move floating-point data between registers.

Syntax

```
mov dst, src
```

where

- dst is the destination register.
- src is a source register.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
mov	x	x	x	x	x	x

Can be used for floating point data. For version vs_1_1, it can also be used to write the address register. When used to update address registers, the values are converted from floating point using rounding to nearest.

The following code fragment shows the operations performed.

```
if(dest is an integer register)
{
    int intSrc = RoundToNearest(src.w);
    dest = intSrc;
}
else
{
    dest = src;
}
```

Related topics

[Vertex Shader Instructions](#)

mov a - VS

Article • 11/20/2019 • 2 minutes to read

Move data from a floating point register to the [Address Register](#), a0.

Syntax

```
mov a dst, src
```

where

- dst must be the [Address Register](#), a0.
- src is a source register.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
mov a		x	x	x	x	x

Moves floating point data to an integer register. The values are converted from floating point using rounding to nearest.

The address register is the only destination register allowed.

The following code fragment shows the operations performed.

```
if(dest is an integer register)
{
    int intSrc = RoundToNearest(src);
    dest = intSrc;
}
else
{
    dest = src;
}
```

For versions 2_x and above, the address register is a component vector. Therefore, any write mask is allowed.

```
mov a0.xz, r0
```

Related topics

[Vertex Shader Instructions](#)

mul - vs

Article • 03/09/2021 • 2 minutes to read

Multiplies sources into the destination.

Syntax

```
mul dst, src0, src1
```

where

- dst is the destination register.
- src0 is a source register.
- src1 is a source register.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
mul	x	x	x	x	x	x

The following code fragment shows the operations performed.

```
dest.x = src0.x * src1.x;  
dest.y = src0.y * src1.y;  
dest.z = src0.z * src1.z;  
dest.w = src0.w * src1.w;
```

Related topics

[Vertex Shader Instructions](#)

nop - VS

Article • 03/09/2021 • 2 minutes to read

No operation is performed.

Syntax

```
nop
```

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
nop	x	x	x	x	x	x

This instruction performs a no-op, or no operation. The syntax for calling it is as follows:

```
nop
```

Related topics

[Vertex Shader Instructions](#)

nrm - VS

Article • 03/09/2021 • 2 minutes to read

Normalize a 3D vector.

Syntax

```
nrm dst, src
```

where

- dst is the destination register.
- src is a source register.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
nrm		x	x	x	x	x

This instruction works conceptually as shown here.

$$\text{squareRootOfTheSum} = (\text{src0.x} * \text{src0.x} + \text{src0.y} * \text{src0.y} + \text{src0.z} * \text{src0.z})^{1/2};$$

```
dest.x = src0.x * (1 / squareRootOfTheSum);
dest.y = src0.y * (1 / squareRootOfTheSum);
dest.z = src0.z * (1 / squareRootOfTheSum);
dest.w = src0.w * (1 / squareRootOfTheSum);
```

The dest and src registers cannot be the same. The dest register must be a temporary register.

This instruction performs the linear interpolation based on the following formula.

```
float f = src0.x*src0.x + src0.y*src0.y + src0.z*src0.z;
if (f != 0)
    f = (float)(1/sqrt(f));
else
    f = FLT_MAX;

dest.x = src0.x*f;
dest.y = src0.y*f;
dest.z = src0.z*f;
dest.w = src0.w*f;
```

Related topics

[Vertex Shader Instructions](#)

pow - VS

Article • 03/09/2021 • 2 minutes to read

Full precision $\text{abs}(\text{src0})^{\text{src1}}$.

Syntax

```
pow dst, src0, src1
```

where

- dst is the destination register.
- src0 is an input source register. Source register requires explicit use of replicate swizzle, that is, exactly one of the .x, .y, .z, .w swizzle components (or the .r, .g, .b, .a equivalents) must be specified.
- src1 is an input source register. Source register requires explicit use of replicate swizzle, that is, exactly one of the .x, .y, .z, .w swizzle components (or the .r, .g, .b, .a equivalents) must be specified.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
pow		x	x	x	x	x

This instruction works as shown here.

```
dest = pow(abs(src0), src1);
```

This is a scalar instruction, therefore the source registers should have replicate swizzles to indicate which channels are used.

The scalar result is replicated to all four output channels.

This instruction could be expanded as $\exp(\text{src1} * \log(\text{src0}))$.

Precision is not lower than 15 bits.

The dest register should be a temporary register, and should not be the same register as src1.

Related topics

[Vertex Shader Instructions](#)

rcp - VS

Article • 03/09/2021 • 2 minutes to read

Computes the reciprocal of the source scalar.

Syntax

```
rcp dst, src
```

where

- dst is the destination register.
- src is a source register. Source register requires explicit use of replicate swizzle, that is, exactly one of the .x, .y, .z, .w swizzle components (or the .r, .g, .b, .a equivalents) must be specified.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
rcp	x	x	x	x	x	x

The following code fragment shows the operations performed.

```
float f = src0;
if(f == 0.0f)
{
    f = FLT_MAX;
}
else
{
    if(f != 1.0)
    {
        f = 1/f;
    }
}

dest = f;
```

The output must be exactly 1.0 if the input is exactly 1.0. A source of 0.0 yields infinity.

Precision should be at least $1.0/(2^{22})$ absolute error over the range (1.0, 2.0) because common implementations will separate mantissa and exponent.

If the source has no subscripts, the x-component is used.

Related topics

[Vertex Shader Instructions](#)

rep - VS

Article • 11/20/2019 • 2 minutes to read

Start a rep...endrep block.

Syntax

```
rep i#
```

where i# is an integer register that specifies the repeat count in the .x component. See [Constant Integer Register](#).

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
rep		x	x	x	x	x

- i#.x specifies the iteration count. The legal range is [0, 255]. Note that this instruction does not increment or decrement the value of i#.x.
- i#.yzw are not used by the repeat block.
- Repeat blocks may be nested. See [Flow Control Nesting Limits](#).
- Repeat blocks are allowed to be either completely inside an if* block or completely surrounding it. No straddling is allowed.
- Using the same i# for different or nested rep instructions is fine - each loop will iterate based on the specified count.

Example

```
rep i2  
    add r0, r0, c0  
endrep
```

Related topics

[Vertex Shader Instructions](#)

ret - vs

Article • 11/20/2019 • 2 minutes to read

Return from a subroutine or a main function.

Syntax

```
ret
```

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
ret		x	x	x	x	x

This instruction takes the address of an instruction from the return address stack and continues execution from it. In the case of the main function, this instruction stops shader execution.

The ret instruction consumes one vertex shader instruction slot.

If a shader contains no subroutines, using ret at the end of the main program is optional.

Multiple return statements are not permitted in the main program or in any subroutine, the first return statement is treated as the end of the main program or subroutine.

Related topics

[Vertex Shader Instructions](#)

rsq - VS

Article • 03/09/2021 • 2 minutes to read

Computes the reciprocal square root (positive only) of the source scalar.

Syntax

```
rsq dst, src
```

where

- dst is the destination register.
- src is a source register. Source register requires explicit use of replicate swizzle, that is, exactly one of the .x, .y, .z, .w swizzle components (or the .r, .g, .b, .a equivalents) must be specified.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
rsq	x	x	x	x	x	x

The following code fragment shows the operations performed.

```
float f = abs(src0);
if (f == 0)
    f = FLT_MAX
else
{
    if (f != 1.0)
        f = 1.0/(float)sqrt(f);
}

dest.z = dest.y = dest.z = dest.w = f;
```

The absolute value is taken before processing.

Precision should be at least $1.0/(2^{22})$ absolute error over the range (1.0, 4.0) because common implementations will separate mantissa and exponent.

If source has no subscripts, the x-component is used. The output must be exactly 1.0 if the input is exactly 1.0. A source of 0.0 yields infinity.

Related topics

[Vertex Shader Instructions](#)

setp_comp - vs

Article • 03/09/2021 • 2 minutes to read

Set the predicate register.

Syntax

```
setp_comp dst, src0, src1
```

Where:

- _comp is a per-channel comparison between the two source registers. It can be one of the following:

Syntax	Comparison
_gt	Greater than
_lt	Less than
_ge	Greater than or equal
_le	Less than or equal
_eq	Equal to
_ne	Not equal to

- dst is the [Predicate Register](#) register, p0.
- src0 is a source register.
- src1 is a source register.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
setp_comp		x	x	x	x	x

This instruction operates as:

```
per channel in destination write mask
{
    dst.channel = src0.channel cmp src1.channel
}
```

For each channel that can be written according to the destination write mask, save the boolean result of the comparison operation between the corresponding channels of src0 and src1 (after the source modifier swizzles have been resolved).

Source registers allow arbitrary component swizzles to be specified.

The destination register allows arbitrary write masks.

The dest register must be the predicate register.

Applying the Predicate Register

Once the predicate register has been initialized with setp, it can be used to control an instruction per component. Here's the syntax:

```
([!]p0[.swizzle]) instruction dest, srcReg, ...
```

Where:

- [!] is an optional boolean NOT
- p0 is the predicate register
- [.swizzle] is an optional swizzle to apply to the contents of the predicate register before using it to mask the instruction. The available swizzles are: .xyzw (default when none specified), or any replicate swizzle: .x/.r, .y/.g, .z/.b or .a/.w.
- instruction is any arithmetic, or texture instruction. This cannot be a static or dynamic flow control instruction.
- dest, srcReg, ... are the registers required by the instruction

Assuming the predicate register has been set up with (true, true, false, false) component values, it can be applied to this instruction:

```
// given r0 = 0,0,1,1  
// given r1 = 1,1,0,0  
setp_le p0, r0, r1  
(p0) add r2, r3, r4
```

to perform a 2 component add.

```
r2.x = r3.x + r4.x  
r2.y = r3.y + r4.y
```

The x and y components of r2 will not be written since the predicate register contained false in components z and w.

Applying the predicate register to an arithmetic or texture instruction increases its instruction slot count by 1.

The predicate register can also be applied to [if pred - vs](#), [callnz pred - vs](#) and [breakp - vs](#) instructions. These flow control instructions do not have any increase in the instruction slot count when using the predicate register.

Related topics

[Vertex Shader Instructions](#)

sge - VS

Article • 11/20/2019 • 2 minutes to read

Computes the sign if the first source is greater than or equal to the second source.

Syntax

```
sge dst, src0, src1
```

where

- dst is the destination register.
- src0 is a source register.
- src1 is a source register.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
sge	x	x	x	x	x	x

The following code fragment shows the operations performed.

```
dest.x = (src0.x >= src1.x) ? 1.0f : 0.0f;  
dest.y = (src0.y >= src1.y) ? 1.0f : 0.0f;  
dest.z = (src0.z >= src1.z) ? 1.0f : 0.0f;  
dest.w = (src0.w >= src1.w) ? 1.0f : 0.0f;
```

Related topics

[Vertex Shader Instructions](#)

sgn - VS

Article • 11/20/2019 • 2 minutes to read

Computes the sign of the input.

Syntax

```
sgn dst, src0, src1, src2
```

where

- dst is the destination register.
- src0 is a source register.
- src1 is a temporary register that holds intermediate results. Following execution, contents are undefined.
- src2 is a temporary register that holds intermediate results. Following execution, contents are undefined.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
sgn		x	x	x	x	x

This instruction works as shown below.

```
for each component in src0
{
    if (src0.component < 0)
        dest.component = -1;
    else
        if (src0.component == 0)
            dest.component = 0;
        else
            dest.component = 1;
}
```

src1 and src2 must be different [Temporary Registers](#).

Related topics

[Vertex Shader Instructions](#)

sincos - VS

Article • 03/09/2021 • 2 minutes to read

Computes sine and cosine, in radians.

Syntax

vs_2_0 and vs_2_x

```
sincos dst.{x|y|xy}, src0.{x|y|z|w}, src1, src2
```

Where:

- dst is the destination register and has to be a [Temporary Register](#) (r#). The destination register must have exactly one of the following three masks: .x | .y | .xy.
- src0 is a source register that provides the input angle, which must be within [-pi, +pi]. {x|y|z|w} is the required replicate swizzle.
- src1 and src2 are source registers and must be two different [Constant Float Register](#) (c#). The values of src1 and src2 must be that of the macros [D3DSINCOSCONST1](#) and [D3DSINCOSCONST2](#) respectively.

vs_3_0

```
sincos dst.{x|y|xy}, src0.{x|y|z|w}
```

Where:

- dst is a destination register and has to be a [Temporary Register](#) (r#). The destination register must have exactly one of the following three masks: .x | .y | .xy.
- src0 is a source register that provides the input angle, which must be within [-pi, +pi]. {x|y|z|w} is the required replicate swizzle.

Remarks

Vertex shader versions

1_1

2_0

2_x

2_sw

3_0

3_sw

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
sincos		x	x	x	x	x

vs_2_0 and vs_2_x Remarks

For vs_2_0 and vs_2_x, sincos can be used with predication, but with one restriction to the swizzle of the [Predicate Register](#) (p0): only replicate swizzle (.x | .y | .z | .w) is allowed.

For vs_2_0 and vs_2_x, the instruction operates as follows: (V = the scalar value from src0 with a replicate swizzle):

- If the write mask is .x:

```
dest.x = cos( V )
dest.y is undefined when the instruction completes
dest.z is undefined when the instruction completes
dest.w is not touched by the instruction
```

- If the write mask is .y:

```
dest.x is undefined when the instruction completes
dest.y = sin( V )
dest.z is undefined when the instruction completes
dest.w is not touched by the instruction
```

- If the write mask is .xy:

```
dest.x = cos( V )
dest.y = sin( V )
dest.z is undefined when the instruction completes
dest.w is not touched by the instruction
```

vs_3_0 Remarks

For vs_3_0, sincos can be used with predication without any restriction. See [Predicate Register](#).

For `vs_3_0`, the instruction operates as follows: (V = the scalar value from `src0` with a replicate swizzle)

- If the write mask is `.x`:

```
dest.x = cos( V )
dest.y is not touched by the instruction
dest.z is not touched by the instruction
dest.w is not touched by the instruction
```

- If the write mask is `.y`:

```
dest.x is not touched by the instruction
dest.y = sin( V )
dest.z is not touched by the instruction
dest.w is not touched by the instruction
```

- If the write mask is `.xy`:

```
dest.x = cos( V )
dest.y = sin( V )
dest.z is not touched by the instruction
dest.w is not touched by the instruction
```

The application can map an arbitrary angle (in radians) to the range $[-\pi, +\pi]$ using the following shader pseudocode:

```
def c0, pi, 0.5, 2*pi, 1/(2*pi)
mad r0.x, input_angle, c0.w, c0.y
frc r0.x, r0.x
mad r0.x, r0.x, c0.z, -c0.x
```

Related topics

[Vertex Shader Instructions](#)

slt - vs

Article • 11/20/2019 • 2 minutes to read

Computes the sign if the first source is less than the second source.

Syntax

```
slt dst, src0, src1
```

where

- dst is the destination register.
- src0 is a source register.
- src1 is a source register.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
slt	x	x	x	x	x	x

The following code fragment shows the operations performed.

```
dest.x = (src0.x < src1.x) ? 1.0f : 0.0f;  
dest.y = (src0.y < src1.y) ? 1.0f : 0.0f;  
dest.z = (src0.z < src1.z) ? 1.0f : 0.0f;  
dest.w = (src0.w < src1.w) ? 1.0f : 0.0f;
```

Related topics

[Vertex Shader Instructions](#)

sub - vs

Article • 03/09/2021 • 2 minutes to read

Subtracts sources.

Syntax

```
sub dst, src0, src1
```

where

- dst is the destination register.
- src0 is a source register.
- src1 is a source register.

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
sub	x	x	x	x	x	x

This instruction performs the subtraction shown in this formula.

```
dest.x = src0.x - src1.x  
dest.y = src0.y - src1.y  
dest.z = src0.z - src1.z  
dest.w = src0.w - src1.w
```

Related topics

[Vertex Shader Instructions](#)

texldl - vs

Article • 03/09/2021 • 2 minutes to read

Sample a texture with a particular sampler. The particular mipmap level-of-detail being sampled has to be specified as the fourth component of the texture coordinate.

Syntax

```
texldl dst, src0, src1
```

Where:

- dst is a destination register.
- src0 is a source register that provides the texture coordinates for the texture sample.
- src1 identifies the source sampler register (s#), where # specifies which texture sampler number to sample. The sampler has associated with it a texture and a control state defined by the [D3DSAMPLERSTATETYPE](#) enumeration (for example, D3DSAMP_MINFILTER).

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
texldl				x	x	

texldl looks up the texture set at the sampler stage referenced by src1. The level-of-detail is selected from src0.w. This value can be negative, in which case the level-of-detail selected is the "zeroth one" (biggest map) with the MAGFILTER. Because src0.w is a floating point value, the fractional value is used to interpolate (if MIPFILTER is LINEAR) between two mip levels. Sampler states MIPMAPLODBIAS and MAXMIPLEVEL are honored. For more information about sampler states, see [D3DSAMPLERSTATETYPE](#).

If a shader program samples from a sampler that does not have a texture set, 0001 is obtained in the destination register.

This is an approximation to the reference device algorithm.

```

LOD = src0.w + LODBIAS;
if (LOD <= 0 )
{
    LOD = 0;
    Filter = MagFilter;
    tex = Lookup( MAX(MAXMIPLEVEL, LOD), Filter );
}
else
{
    Filter = MinFilter;
    LOD = MAX( MAXMIPLEVEL, LOD);
    tex = Lookup( Floor(LOD), Filter );
    if( MipFilter == LINEAR )
    {
        tex1 = Lookup( Ceil(LOD), Filter );
        tex = (1 - frac(src0.w))*tex + frac(src0.w)*tex1;
    }
}

```

Restrictions:

- The texture coordinates should not be scaled by texture size.
- dst must be a [Temporary Register](#) (r#).
- dst can accept a write mask. See [Destination Register Masking](#).
- Defaults for missing components are either 0 or 1, and depend on the texture format.
- src1 must be a [Sampler \(Direct3D 9 asm-vs\)](#) (s#). src1 may not use a negate modifier. src1 may use swizzle, which is applied after sampling before the write mask is honored. The sampler must have been declared (using [dcl_samplerType \(sm3 - vs asm\)](#)) at the beginning of the shader.
- The number of coordinates required to perform the texture sample depends on how the sampler was declared. If it was declared as a cube, a three-component texture coordinate is required (.rgb). Validation enforces that coordinates provided to texldl are sufficient for the texture dimension declared for the sampler. However, it is not guaranteed that the application actually set a texture (through the API) with dimensions equal to the dimension declared for the sampler. In such a case, the runtime will attempt to detect mismatches (possibly in debug only). Sampling a texture with fewer dimensions than are present in the texture coordinate will be allowed, and will be assumed to ignore the extra texture coordinate components. Conversely, sampling a texture with more dimensions than are present in the texture coordinate is not allowed.
- If the src0 (texture coordinate) is a [Temporary Register](#) (r#), the components required for the lookup (described above) must have been previously written.

- Sampling unsigned RGB textures will result in float values between 0.0 and 1.0.
- Sampling signed textures will result in float values between -1.0 to 1.0.
- When sampling floating point textures, Float16 means that the data will range within MAX_FLOAT16. Float32 means the maximum range of the pipeline will be used. Sampling outside of either range is undefined.
- There is no dependent read limit.

Related topics

[Vertex Shader Instructions](#)

VS

Article • 11/20/2019 • 2 minutes to read

This instruction specifies the shader version number. This instruction works on all shader versions.

Syntax

```
vs_mainVer_subVer
```

Input Arguments

Input arguments contain a single main version number with a single sub version number. The allowable combinations are listed in the table below.

Main versions	Sub versions
1	1
2	0, sw (software), x (extended)
3	0, sw (software)

Remarks

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
vs	x	x	x	x	x	x

This instruction must be the first non-comment instruction in a vertex shader.

This instruction is supported in all vertex shader versions.

Hardware accelerated versions of the software (versions without _sw in the version number), can process vertices with hardware acceleration or use software vertex

processing. Software versions (versions with _sw in the version number) process vertices only with software.

Examples

This partial example declares a version 1_1 vertex shader.

```
vs_1_1
```

This partial example declares a version 2 software vertex shader.

```
vs_2_sw
```

Related topics

[Vertex Shader Instructions](#)

Vertex Shader Registers

Article • 08/23/2019 • 2 minutes to read

- [Registers - vs_1_1](#)
- [Registers - vs_2_0](#)
- [Registers - vs_2_x](#)
- [Registers - vs_3_0](#)
- [Vertex Shader Register Modifiers](#)

Related topics

[Vertex Shaders](#)

Vertex Shader Register Modifiers

Article • 08/23/2019 • 2 minutes to read

- [Vertex Shader Source Register Modifiers](#)
- [Source Register Swizzling](#)
- [Destination Register Masking](#)

Related topics

[Vertex Shader Registers](#)

Vertex Shader Source Register Modifiers

Article • 08/23/2019 • 2 minutes to read

Source modifiers can be applied to modify the data read from a source register before the data is used by the instruction.

Negate

Negate the contents of the source register.

Component modifier	Description
- r	Source negation

The negate modifier cannot be used on second source register of these instructions: [m3x2 - vs](#), [m3x3 - vs](#), [m3x4 - vs](#), [m4x3 - vs](#), [m4x4 - vs](#).

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
-	x	x	x	x	x	x

Absolute Value

Take the absolute value of the register.

Vertex shader versions	1_1	2_0	2_x	2_sw	3_0	3_sw
abs					x	x

If any version 3 shader reads from one or more constant float registers (c#), one of the following must be true.

- All of the constant floating-point registers must use the abs modifier.
- None of the constant floating-point registers can use the abs modifier.

Related topics

[Vertex Shader Register Modifiers](#)

Source Register Swizzling (HLSL VS reference)

Article • 11/23/2019 • 2 minutes to read

Before an instruction runs, the data in a source register is copied to a temporary register. Swizzling refers to the ability to copy any source register component to any temporary register component. Swizzling does not affect the source register data.

Component Swizzling

As shown in the following table, swizzling can be applied to the individual components of the source register data (where r is one of the valid vertex shader input [Registers - vs_1_1](#)).

Component modifier	Description
r.[xyzw][xyzw][xyzw][xyzw]	Source swizzle

- All four components are always copied. If fewer than four components are specified, the last component is repeated (xy means .yyyy). If no components are specified, x is repeated (.xxxx).
- The components can appear in any order. v0.ywx results in v0.ywxx.
- The rgba components can be used respectively for xyzw (r for x, g for b, etc.).
- These instructions implement source-register single component swizzles: exp, expp, log, logp, pow, rcp, rsq. The result of these instructions is copied to all four destination register components.

Swizzling cannot be used on the [m3x2 - vs](#), [m3x3 - vs](#), [m4x3 - vs](#), and [m4x4 - vs](#) instructions.

Related topics

[Vertex Shader Register Modifiers](#)

Destination Register Masking

Article • 08/23/2019 • 2 minutes to read

Masking specifies which components of the destination register will be updated with the result of an instruction. Texture registers have one set of rules and nontexture registers have another set of rules.

- `dx9_graphics_reference_asm_vs_registers_modifiers_masking` - This section contains rules for applying masks to destination registers.
- `Texture_Register_Masks` - Texture registers have some unique rules.

Destination Register Masking

As shown in the following table, masking (where r is one of the valid vertex shader [Vertex Shader Registers](#)) can be applied to the individual components of the vector data.

Component modifier	Description
<code>r.{x}{y}{z}{w}</code>	Destination mask

- In general, specifying destination register write masks is good coding style. It makes code easier to read and maintain.
- Any combination of components may be specified (including none) as long as x precedes y, y precedes z, and z precedes w.
- The oPts and oFog output registers must use only one mask.
- Certain instructions require the destination register to use a single write mask: exp, expp, log, logp, pow, rcp, and rsq.
- In version 1.0, the frc instruction required one of the following mask combinations: .x or .y or .xy. Version 2.0 has no mask restriction.
- sincos requires one of the following mask combinations: .x or .y or .xyz.
- m3x2 requires the .xy write mask.
- m3x3 and m4x3 require the .xyz write mask.
- m3x4 and m4x4 require either the .xyz write mask or the default write mask (xyzw).

Texture Register Masks

The validation rules for using modifiers on texture coordinate registers are tighter than the validation rules for other registers.

- If oTn is written, all previous registers ($oTn-1 \sim oT0$) have to be written as well.
- The "combined" write mask for any $oT\#$ register must be exactly one of the following:
 - .x
 - .xy
 - .xyz
 - .xyzw (which is equivalent to not using any component modifiers)

For example, a vertex shader can output to texture registers in separate instructions.

```
oT1.y  
oT0.y  
oT2  
oT0.xz  
oT1.x
```

Or, the instructions can be combined.

```
oT0.xyz  
oT1.xy  
oT2.xyzw
```

These restrictions apply only to the texture coordinate registers.

Related topics

[Vertex Shader Register Modifiers](#)

Registers - vs_1_1

Article • 08/23/2019 • 2 minutes to read

This section contains reference information for the input and output registers implemented by vertex shader version 1_1.

Input Registers

Register	Name	Count	R/W	# Read ports	# Reads / inst	Dimension	RelAddr	Defaults	Requires DCL
									inst
a0	Address Register	1	R/W	1	Unlimited	See note 3	No	None	No
c#	Constant Float Register	See note 2	R	1	Unlimited	4	a0.x	(0, 0, 0, 0)	No
v#	Input Register	16	R	1	Unlimited	4	No	See note 1	Yes
r#	Temporary Register	12	R/W	3	Unlimited	4	No	None	No

Notes:

1. Partial (0, 0, 0, 1) - If only a subset of channels are updated, the remaining channels will default to (0, 0, 0, 1).
2. Equal to D3DCAPS9.MaxVertexShaderConst (at least 96 for vs_1_1).
3. Only .x channel is available.

Output Registers

Register	Name	Count	R/W	Dimension	RelAddr	Defaults	Requires DCL
oPos	Position Register	1	W	4	No	None	No
oFog	Fog Register	1	W	1	No	None	No
oPts	Point Size Register	1	W	1	No	None	No

Register	Name	Count	R/W	Dimension	RelAddr	Defaults	Requires DCL
oD#	Color Register; See note 1	2	W	4	No	None	No
oT#	Texture Coordinate Register	8	W	4	No	None	No

Notes:

- oD0 is the diffuse color output; oD1 is the specular color output.

Related topics

[Vertex Shader Registers](#)

Registers - vs_2_0

Article • 08/23/2019 • 2 minutes to read

This section contains reference information for the input and output registers implemented by vertex shader version 2_0.

Input Registers

Register	Name	Count	R/W	# Read ports	# Reads / inst	Dimension	RelAddr	Defaults	Requires DCL
v#	Input Register	16	R	1	Unlimited	4	No	See note 1	Yes
r#	Temporary Register	12	R/W	3	Unlimited	4	No	None	No
c#	Constant Float Register	See note 2	R	1	2	4	a0 / aL	(0, 0, 0, 0)	No
a0	Address Register	1	R/W	1	2	4	No	None	No
b#	Constant Boolean Register	16	R	1	1	1	No	FALSE	No
i#	Constant Integer Register	16	R	1	1	4	No	(0, 0, 0, 0)	No
aL	Loop Counter Register	1	R	1	2	1	No	None	No

Notes:

1. Partial (0, 0, 0, 1) - If only a subset of channels are updated, the remaining channels will default to (0, 0, 0, 1).
2. Equal to D3DCAPS9.MaxVertexShaderConst (at least 256 for vs_2_0).

Output Registers

Register	Name	Count	R/W	Dimension	RelAddr	Defaults	Requires DCL
oPos	Position Register	1	W	4	No	None	No
oFog	Fog Register	1	W	1	No	None	No
oPts	Point Size Register	1	W	1	No	None	No
oD#	Color Register; See note 1	2	W	4	No	None	No
oT#	Texture Coordinate Register	8	W	4	No	None	No

Notes:

- oD0 is the diffuse color output; oD1 is the specular color output.

Related topics

[Vertex Shader Registers](#)

Registers - vs_2_x

Article • 08/23/2019 • 2 minutes to read

This section contains reference information for the input and output registers implemented by vertex shader version 2_x.

Input Registers

Register	Name	Count	R/W	# Read ports	# inst	Dimension	RelAddr	Defaults	Requires DCL
v#	Input Register	16	R	1	Unlimited	4	No	See note 1	Yes
r#	Temporary Register	See note 2	R/W	3	Unlimited	4	No	None	No
c#	Constant Float Register	See note 3	R	1	2	4	a0 / aL	(0, 0, 0, 0)	No
a0	Address Register	1	R/W	1	2	4	No	None	No
b#	Constant Boolean Register	16	R	1	1	1	No	FALSE	No
i#	Constant Integer Register	16	R	1	1	4	No	(0, 0, 0, 0)	No
aL	Loop Counter Register	1	R	1	2	1	No	None	No
p0	Predicate Register	1	R/W	1	1	4	No	None	No

Notes:

1. Partial (0, 0, 0, 1) - If only a subset of channels are updated, the remaining channels will default to (0, 0, 0, 1).
2. Equal to D3DCAPS9.VS20Caps.NumTemps (at least 12 for vs_2_x).

3. Equal to D3DCAPS9.MaxVertexShaderConst (at least 256 for vs_2_x).

Output Registers

Register	Name	Count	R/W	Dimension	RelAddr	Defaults	Requires DCL
oPos	Position Register	1	W	4	No	None	No
oFog	Fog Register	1	W	1	No	None	No
oPts	Point Size Register	1	W	1	No	None	No
oD#	Color Register ; See note 1	2	W	4	No	None	No
oT#	Texture Coordinate Register	8	W	4	No	None	No

Notes:

- oD0 is the diffuse color output; oD1 is the specular color output.

Related topics

[Vertex Shader Registers](#)

Registers - vs_3_0

Article • 08/23/2019 • 2 minutes to read

This section contains reference information for the input and output registers implemented by vertex shader version 3_0.

Input Registers

Register	Name	Count	R/W	# Read ports	# Reads / inst	Dimension	RelAddr	Defaults	Requires DCL
v#	Input Register	16	R	1	Unlimited	4	a0/aL	See note 1	Yes
r#	Temporary Register	32	R/W	3	Unlimited	4	No	None	No
c#	Constant Float Register	See note 2	R	1	Unlimited	4	a0/aL	(0, 0, 0, 0)	No
a0	Address Register	1	R/W	1	Unlimited	4	No	None	No
b#	Constant Boolean Register	16	R	1	1	1	No	FALSE	No
i#	Constant Integer Register	16	R	1	1	4	No	(0, 0, 0, 0)	No
aL	Loop Counter Register	1	R	1	Unlimited	1	No	None	No
p0	Predicate Register	1	R/W	1	1	4	no	none	no
s#	Sampler (Direct3D 9 asm-vs)	4	R	1	1	4	No	See note 3	Yes

Notes:

1. Partial (0, 0, 0, 1) - If only a subset of channels are updated, the remaining channels will default to (0, 0, 0, 1).
2. Equal to D3DCAPS9.MaxVertexShaderConst (at least 256 for vs_3_0).
3. Defaults for sampler lookup exist, but values depend on texture format.

Output Registers

Output registers have been collapsed into 12 o# (output) registers. These can be used for anything the user wants to interpolate for the pixel shader: texture coordinates, colors, fog, etc.

Register	Name	Count	R/W	Dimension	RelAddr	Defaults	Requires DCL
o#	Output Register	12	W	4	aL	None	Yes

Related topics

[Vertex Shader Registers](#)

Relative Addressing (HLSL VS reference)

Article • 06/18/2021 • 2 minutes to read

The [] syntax can be used only in register types that can be relatively addressed in certain shader models. The supported forms of [] syntax are listed as follows:

Where:

- "R" denotes any register type that can be relatively addressed.
- "A" denotes any register that can be used as an index to relatively address other registers.
- $n_0 - n_i$, $m_0 - m_j$, and k are integers ≥ 0 .

[] syntax	Effective index	Examples
$R[A + m_0 + \dots + m_j]$	$A + m_0 + \dots + m_j$	$c[a0.x + 3 + 7]$
$R[k] (= Rk)$	k	$c[10] (= c10)$
$R[A]$	A	$c[a0.y]$
$Rk[n_0 + \dots + n_i + A + m_0 + \dots + m_j]$	$A + k + n_0 + \dots + n_i + m_0 + \dots + m_j$	$c8[3 + 2 + a0.w + 5 + 6 + 1]$
$R[n_0 + \dots + n_i + A + m_0 + \dots + m_j]$	$A + n_0 + \dots + n_i + m_0 + \dots + m_j$	$c[2 + 1 + aL + 3 + 4 + 5]$
$Rk[A]$	$A + k$	$c12[aL], c0[a0.z]$
$Rk[A + m_0 + \dots + m_j]$	$A + k + m_0 + \dots + m_j$	$v1[aL + 4 + 8]$
$R[n_0 + \dots + n_i + A]$	$A + n_0 + \dots + n_i$	$o[3 + 1 + aL]$
$Rk[n_0 + \dots + n_i + A]$	$A + k + n_0 + \dots + n_i$	$o1[2 + 1 + 3 + aL]$

The registers are available in the following versions:

Register type	Vertex Shader Versions
a0	All
aL	vs_2_0 and higher
cn	vs_1_1 and higher
on	vs_3_0

Related topics

[Vertex Shader Registers](#)

Address Register

Article • 08/23/2019 • 2 minutes to read

The a0 register is an address register. A single register is available in version vs_1_1. The address register, designated as a0.x in vs_1_1, can be used as a signed integer offset for relative addressing into the constant register file. For versions vs_2_0 and above, all four components (.x, .y, .z, .w) are available for relative addressing.

```
c[a0.x + n]
```

The address register cannot be read by a vertex shader, it can only be used for relative addressing of a constant register. Reading values outside of the legal range will return (0.0, 0.0, 0.0, 0.0). The address register can only be a destination for the [mov - vs](#) instruction. If a floating-point number is moved into an integer register, a round-to-nearest conversion happens.

All shaders must initialize the address register before using it. For version vs_2_0 and above, the [mova - vs](#) instruction can move a floating-point value to an address register.

Vertex shader versions	1_1	2_0	2_sw	2_x	3_0	3_sw
Address Register	x	x	x	x	x	x

Related topics

[Vertex Shader Registers](#)

Constant float register (HLSL VS reference)

Article • 08/19/2020 • 2 minutes to read

Vertex shader input register for a four component floating-point constant. Set a constant register with either [def - vs](#) or [SetVertexShaderConstantF](#).

The constant register file is read-only from the perspective of the vertex shader. Any single instruction may access only one constant register. However, each source in that instruction may independently swizzle and negate that vector as it is read.

The behavior of shader constants has changed between Direct3D 8 and Direct3D 9.

- For Direct3D 9, constants set with defx assign values to the shader constant space. The lifetime of a constant declared with defx is confined to the execution of that shader only. Conversely, constants set using the APIs SetXXXShaderConstantX initialize constants in global space. Constants in global space are not copied to local space (visible to the shader) until SetxxxShaderConstants is called.
- For Direct3D 8, constants set with defx or the APIs both assign values to the shader constant space. Each time the shader is executed, the constants are used by the current shader regardless of the technique used to set them.

A constant register is designated as either absolute or relative:

```
c[n]          ; absolute  
c[a0.x + n]  ; relative - supported only in version 1_1
```

The constant register can be read, therefore, either by using an absolute index or with a relative index from an address register. Reads from out-of-range registers return (0.0, 0.0, 0.0, 0.0).

Examples

Here is an example declaring two floating-point constants within a shader.

```
def c40, 0.0f,0.0f,0.0f,0.0f;
```

These constants are loaded every time [SetVertexShader](#) is called.

Here is an example using the API.

```
// Set up the vertex shader constants.  
{  
    D3DXMATRIXA16 mat;  
    D3DXMatrixMultiply( &mat, &m_matView, &m_matProj );  
    D3DXMatrixTranspose( &mat, &mat );  
  
    D3DXVECTOR4 vA( sinf(m_fTime)*15.0f, 0.0f, 0.5f, 1.0f );  
    D3DXVECTOR4 vD( D3DX_PI, 1.0f/(2.0f*D3DX_PI), 2.0f*D3DX_PI, 0.05f );  
  
    // Taylor series coefficients for sin and cos.  
    D3DXVECTOR4 vSin( 1.0f, -1.0f/6.0f, 1.0f/120.0f, -1.0f/5040.0f );  
    D3DXVECTOR4 vCos( 1.0f, -1.0f/2.0f, 1.0f/ 24.0f, -1.0f/ 720.0f );  
  
    m_pd3dDevice->SetVertexShaderConstantF( 0, (float*)&mat, 4 );  
    m_pd3dDevice->SetVertexShaderConstantF( 4, (float*)&vA, 1 );  
    m_pd3dDevice->SetVertexShaderConstantF( 7, (float*)&vD, 1 );  
    m_pd3dDevice->SetVertexShaderConstantF( 10, (float*)&vSin, 1 );  
    m_pd3dDevice->SetVertexShaderConstantF( 11, (float*)&vCos, 1 );  
}
```

If you are setting constant values with the API, there is no shader declaration required.

Vertex shader versions	1_1	2_0	2_sw	2_x	3_0	3_sw
Constant Register	x	x	x	x	x	x

Related topics

[Vertex Shader Registers](#)

Constant Integer Register (HLSL VS reference)

Article • 08/19/2020 • 2 minutes to read

Constant integer registers are used only by [loop - vs](#) and [rep - vs](#).

They can be set using [defi - vs](#) or [SetVertexShaderConstantI](#).

When used as an argument to the [loop - vs](#) instruction:

- .x is the iteration count. ([rep - vs](#) uses only this component).
- .y is the initial value for the loop counter.
- .z is the increment step for the loop counter.

The behavior of shader constants has changed between Direct3D 8 and Direct3D 9.

- For Direct3D 9, constants set with defx assign values to the shader constant space. The lifetime of a constant declared with defx is confined to the execution of that shader only. Conversely, constants set using the APIs SetXXXShaderConstantX initialize constants in global space. Constants in global space are not copied to local space (visible to the shader) until SetxxxShaderConstants is called.
- For Direct3D 8, constants set with defx or the APIs both assign values to the shader constant space. Each time the shader is executed, the constants are used by the current shader regardless of the technique used to set them.

Related topics

[Vertex Shader Registers](#)

Constant Boolean register (HLSL VS reference)

Article • 08/19/2020 • 2 minutes to read

This register is a collection of bits used in static flow control instructions (for example, [if](#) [bool - vs - else - vs - endif - vs](#)). There are 16 of them, therefore, a shader can have 16 independent branch conditions. They can be set using [defb - vs](#) or [SetVertexShaderConstantB](#).

The behavior of shader constants has changed between Direct3D 8 and Direct3D 9.

- For Direct3D 9, constants set with defx assign values to the shader constant space. The lifetime of a constant declared with defx is confined to the execution of that shader only. Conversely, constants set using the APIs SetXXXShaderConstantX initialize constants in global space. Constants in global space are not copied to local space (visible to the shader) until SetxxxShaderConstants is called.
- For Direct3D 8, constants set with defx or the APIs both assign values to the shader constant space. Each time the shader is executed, the constants are used by the current shader regardless of the technique used to set them.

Related topics

[Vertex Shader Registers](#)

Color Register

Article • 02/04/2021 • 2 minutes to read

These vertex shader output registers contain a color value.

Register	Description
oD0	diffuse color register.
oD1	specular color register.

The oD0 value is interpolated and is written to the pixel shader color register 0 (v0). The oD1 value is interpolated and written to the pixel shader color register 1 (v1). For more information about pixel shader color registers, see the pixel shader [Input Color Register](#) topic.

Remarks

Example

```
min oD0, r0, c1.x
```

Related topics

[Vertex Shader Registers](#)

Fog Register

Article • 08/23/2019 • 2 minutes to read

This vertex shader output register contains a per-vertex fog color.

A register consists of properties that determine how each register behaves.

Property	Description
Name	<code>oFog</code>
Count	One vector, of which only one component can be used and must be specified by the component mask
I/O permissions	Write-only.

The output fog value registers. The value is the fog factor to be interpolated and then routed to the fog table. Only the scalar x-component of the fog is used.

Related topics

[Vertex Shader Registers](#)

Input Register

Article • 08/23/2019 • 2 minutes to read

Vertex shader input register.

Data from each vertex (using one or more input vertex streams) is loaded into the vertex input registers before the vertex shader is run. The input registers consist of 16 four-component floating-point vectors, designated as v0 through v15. These registers are read-only. An input register is bound to vertex data through a vertex declaration.

The following register properties control how each register behaves:

Property	Description
Name	v[n] - n is an optional register number. 0 is the default value used, if it is omitted.
Count	A maximum of 16 registers, v0 - v15.
I/O permissions	Read-only. This register cannot be written by the API or within the shader.
Read ports	1. This is the number of times a register can be read within a single instruction. See below.

Any single instruction can access only one vertex input register. However, each source in the instruction can independently swizzle and negate that vector as it is read.

Example

Here is an example using a vertex declaration to bind 2D vertex position data to register v0.

The vertex declaration belongs in the application:

```
D3DVERTEXELEMENT9 decl[] =
{
    { 0, 0, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT,
D3DDECLUSAGE_POSITION, 0 },
    D3DDECL_END()
};
```

Here is the corresponding vertex shader declaration:

```
dcl_position v0
```

Vertex shader versions	1_1	2_0	2_sw	2_x	3_0	3_sw
Position Register	x	x	x	x	x	x

Related topics

[Vertex Shader Registers](#)

Loop Counter Register (HSL VS reference)

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

The only register in this bank is the current loop counter (aL) register. It automatically gets incremented in each execution of the [loop - vs...endloop - vs](#) block. So it can be used in the block for relative addressing if needed and is invalid to use it outside the loop.

Related topics

[Vertex Shader Registers](#)

Output Registers

Article • 08/23/2019 • 2 minutes to read

- Vertex Color Register
- Fog Register
- Position_Register
- Point_Size_Register
- Texture_Coordinate_Register

Register names are preceded by a lowercase letter o, indicating that the output registers are write-only.

Vertex Color Register - oD0, oD1

oD0 is the diffuse color register. oD1 is the specular color register. The oD0 value is interpolated and is written to the input color register 0 (v0) of the pixel shader. The oD1 value is interpolated and written to the input color register 1 (v1) of the pixel shader. For more information about pixel shader color registers, see Registers.

Vertex shader versions	1_1	2_0	2_sw	2_x	3_0	3_sw
Vertex Color Register	x	x	x		x	

Fog Register - oFog

The output fog value registers. The value is the fog factor to be interpolated and then routed to the fog table. Only the scalar x-component of the fog is used. Values are clamped between zero and one before passing to the rasterizer.

Vertex shader versions	1_1	2_0	2_sw	2_x	3_0	3_sw
Fog Register	x	x	x		x	

Position Register - oPos

The output position registers. The value is the position in homogeneous clipping space. This value must be written by the vertex shader.

Vertex shader versions	1_1	2_0	2_sw	2_x	3_0	3_sw
Position Register	x	x	x	x		

Point Size Register - oPts

The output point-size registers. Only the scalar x-component of the point size is used.

Vertex shader versions	1_1	2_0	2_sw	2_x	3_0	3_sw
Point Size Register	x	x	x	x		

Texture Coordinate Register - oT0 to oT7

The output texture coordinates registers. Specifically, these are an array of output data registers that are iterated and used as texture coordinates by the texture sampling stages routing data to the pixel shader.

Vertex shader versions	1_1	2_0	2_sw	2_x	3_0	3_sw
Texture Coordinate Register	x	x	x	x		

When writing to a texture coordinate register, it is recommended to pass only as many floating point values as the dimension of the corresponding texture map. Control the values passed with a modifier. For example, use .xy for a 2D texture map.

When texture projection is enabled for a texture stage, all four floating point values must be written to the corresponding texture register.

Any of the D3DTTFF* texture transform flags should be zero when the programmable pipeline is being used.

Texture Coordinate Range

Object vertex data supplies input texture coordinates. Objects that do not use tiled textures commonly have texture coordinates in the range [0,1]. Objects that use tiled textures, such as terrain, typically have texture coordinates that range from [-?, +?] where ? can be a large floating point number.

Texture coordinate interpolation is performed on vertex data for rasterization. During rasterization, texture coordinates are interpolated between object vertices, modified by texture wrapping and scaled by the texture size (also taking into account texture address mode) to produce an integer index. The index is then used to perform a texture lookup. MaxTextureRepeat can be used to determine how many times a texture can be tiled.

If texture coordinates are read directly into a pixel shader (using texcoord or texcrd), the texture coordinate range depends on the instruction and the pixel shader version.

Related topics

[Vertex Shader Registers](#)

Point Size Register

Article • 08/23/2019 • 2 minutes to read

This vertex shader output register contains per-vertex point size data.

Vertex shader versions	1_1	2_0	2_sw	2_x	3_0	3_sw
Point Size Register	x	x	x	x	x	x

A register consists of properties that determine how each register behaves.

Property	Description
Name	oPts
Count	1 vector, of which only 1 component can be used and must be specified by the component mask.
I/O permissions	Write-only.

Only the scalar x-component of the point size is used.

Related topics

[Vertex Shader Registers](#)

Position Register

Article • 08/23/2019 • 2 minutes to read

This vertex shader output register contains per-vertex position data.

Vertex shader versions	1_1	2_0	2_sw	2_x	3_0	3_sw
Position Register	x	x	x	x	x	x

A register consists of properties that determine how each register behaves.

Property	Description
Name	oPos
Count	1 vector
I/O permissions	Write-only.

The value is the position in homogeneous clipping space. This value must be written by the vertex shader.

Example

```
dcl_position v0

def c40, 0.0f,0.0f,0.0f,0.0f;
// transform into projection space
m4x4 r0,v0,c8
max r0.z,c40.z,r0.z //clamp to 0
max r0.w,c12.x,r0.w //clamp to near clip plane
mov oPos,r0
```

Related topics

[Vertex Shader Registers](#)

Predicate Register (HLSL VS reference)

Article • 11/23/2019 • 2 minutes to read

This vertex shader output register contains a per-channel Boolean value.

A predicate register is supported by the following versions.

Vertex shader versions	1_1	2_0	2_sw	2_x	3_0	3_sw
predicate register				x	x	x

Here are the register properties.

Register type	Count	R/W	# Read ports	# Reads/inst	Dimension	RelAddr	Defaults	Requires DCL
Predicate(p)	1	R/W	1	1	4	N/A	None	N

The predicate register can be modified with [setp_comp - vs](#). There are no default values for this register, an application needs to set the register before it is used.

Related topics

[Vertex Shader Registers](#)

Sampler (Direct3D 9 asm-vs)

Article • 08/19/2020 • 2 minutes to read

A sampler is a input pseudo-register for a vertex shader, which is used to identify the sampling stage. There are four vertex shader samplers: s0 to s3. Four texture surfaces can be read in a single shader pass.

Sampler (Direct3D 9 asm-vs)s are pseudo registers because you cannot directly read or write to them.

A sampling unit corresponds to the texture sampling stage, encapsulating the sampling-specific state provided by [SetSamplerState](#). Each sampler uniquely identifies a single texture surface, which is set to the corresponding sampler using the [SetTexture](#). However, the same texture surface can be set at multiple samplers.

At draw time, a texture cannot be simultaneously set as a render target and a texture at a stage.

Because there are four samplers, up to four texture surfaces can be read from in a single shader pass. A sampler might appear as the only argument in the texture load instruction: [texldl - vs](#).

In vs_3_0, if a sampler is used, it needs to be declared at the beginning of the shader program using the [dcl_samplerType \(sm3 - vs asm\)](#) instruction.

Vertex shader versions	1_1	2_0	2_sw	2_x	3_0	3_sw
Sampler				x	x	

Related topics

[Vertex Shader Registers](#)

[Vertex Textures in vs_3_0 \(DirectX HLSL\)](#)

Temporary Register (HLSL VS reference)

Article • 11/23/2019 • 2 minutes to read

A vertex shader temporary register is used to hold intermediate results.

A temporary register must be initialized before it is used. Each temporary register has single-write and triple-read access. This means that a single shader instruction can use as many as three temporary registers as inputs.

Values in a temporary register that remain from preceding invocations of the vertex shader cannot be used.

A register consists of properties that determine how each register behaves.

Property	Description
Name	r[n]. n is an optional register number. The default is 0, and is the value used if no value is specified.
Count	A maximum of 12 registers.
I/O permissions	Read/write. This register can be read or written by the API or by the shader.
Read ports	The number of times a register can be read within a single instruction is 3. A temporary register is the only register that can be read and written more than once in a single instruction.

Each temporary register has single-write and triple-read access. Therefore, an instruction can have as many as three temporary registers in its set of input source operands.

No values in a temporary register that remain from preceding invocations of the vertex shader can be used. Vertex shaders that read a value from a temporary register before writing to it will fail the Direct3D API call to create the vertex shader.

Example

Here is an example using a temporary register:

```
def c4, 0,0,0,1  
...
```

```

// Decompress position
mov r0.x, v0.x
mov r0.y, c4.w      // 1
mov r0.z, v0.y
mov r0.w, c4.w      // 1

// Compute theta from distance and time
mov r4.xz, r0        // xz

```

Vertex shader versions	1_1	2_0	2_sw	2_x	3_0	3_sw
Temporary Register	x	x	x	x	x	x

Related topics

[Vertex Shader Registers](#)

Texture Coordinate Register (HLSL VS reference)

Article • 08/19/2020 • 2 minutes to read

This vertex shader output register contains per-vertex texture coordinates.

A register consists of properties that determine how each register behaves.

Property	Description
Name	oT0 - oT7
Count	Eight vectors
I/O permissions	Write-only

The output texture coordinates registers are an array of output data registers. The register data is iterated and used as texture coordinates by the texture sampling stages to supply data to the pixel shader.

When writing to a texture coordinate register, it is recommended that you pass only as many floating point values as the dimension of the corresponding texture map. Control the values that are passed with a modifier. For example, use .xy for a 2D texture map.

The fixed function vertex pipeline flags, [D3DTEXTURETRANSFORMFLAGS](#) (D3DTTFF_COUNT1, D3DTTFF_COUNT2, D3DTTFF_COUNT3, D3DTTFF_COUNT4), should be set to zero if you are using a programmable vertex shader.

Object vertex data supplies input texture coordinates. Objects that do not use tiled textures commonly have texture coordinates in the range [0,1]. Objects that use tiled textures, such as terrain, typically have texture coordinates that range from [-n,+n] where n can be any floating point number.

Texture coordinate interpolation is performed on vertex data for rasterization. During rasterization, texture coordinates are interpolated between object vertices, modified by texture wrapping, and scaled by the texture size (also taking into account texture addressing modes) to produce an integer index. The index is then used to perform a texture lookup. Use the MaxTextureRepeat value in [D3DCAPS9](#) to determine how many times a texture can be tiled.

Example

Declare the texture coordinate register.

```
dcl_texcoord v7
```

Copy the per-vertex texture coordinates to the output register.

```
mov oT0, v7
```

Vertex shader versions	1_1	2_0	2_sw	2_x	3_0	3_sw
Texture Coordinate Register	x	x	x	x	x	x

Related topics

[Vertex Shader Registers](#)

Pixel Shaders

Article • 08/23/2019 • 2 minutes to read

- [ps_1_1, ps_1_2, ps_1_3, ps_1_4](#)
- [ps_2_0](#)
- [ps_2_x](#)
- [ps_3_0](#)
- [Registers](#)
- [Pixel Shader Instructions](#)
- [Pixel Shader Differences](#)

Related topics

[Asm Shader Reference](#)

ps_1_1, ps_1_2, ps_1_3, ps_1_4

Article • 08/23/2019 • 2 minutes to read

The pixel shader assembler is made up of a set of instructions that operate on pixel data contained in registers. Operations are expressed as instructions comprised of an operator and one or more operands. Instructions use registers to transfer data in and out of the pixel shader ALU. Registers can also be used by some instructions to hold temporary results.

ⓘ Note

HLSL support for pixel shader 1.x is deprecated.

Instructions

There are two main categories of pixel shader instructions: arithmetic instructions and texture addressing instructions. Arithmetic instructions modify color data. Texture addressing operations process texture coordinate data and in most cases, sample a texture. Pixel shader instructions are run on a per-pixel basis; that is, they have no knowledge of other pixels in the pipeline.

Texture addressing instructions each consume one slot, but arithmetic instructions can be paired to enable both color components (RGB) and an alpha component instruction in a single slot.

[ps_1_1, ps_1_2, ps_1_3, ps_1_4 Instructions](#) contains a list of the available instructions.

When multisampling is enabled, pixel shaders only get run once per pixel, not once for every subpixel. Multisampling only increases the resolution of polygon edges, as well as depth and stencil tests. For example, if 3x3 multisampling is enabled, and a triangle being rasterized is found to cover five of the nine subpixels for a particular pixel, the pixel shader gets run once and the same color result is applied to all five subpixels.

Registers

[ps_1_1_ps_1_2_ps_1_3_ps_1_4 Registers](#) lists the different registers used by the shader ALU.

Modifiers

Modifiers for `ps_1_X` can be used to change the functionality of an instruction, or the data read from or written to a register.

Direct3D 9 requires intermediate computations to maintain at least 8-bit precision for all surface formats. Both higher precision (12-bit) for in-stage math, and saturation to 8-bits between texture stages are recommended. No modifiable rounding modes or exceptions are supported. Multiplication should be supported with a round-to-nearest precision to keep precision loss to a minimum.

Sampler Count

The number of texture samplers available is:

- For `ps_1_0` - `ps_1_3`, the maximum is 4.
- For `ps_1_4`, the maximum is 6.

Related topics

[Pixel Shaders](#)

ps_2_0

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

A programmable pixel shader is made up of a set of instructions that operate on pixel data. Registers transfer data in and out of the ALU. Additional control can be applied to modify the instruction, the results, or what data gets written out.

- [ps_2_0 Instructions](#) contains a list of the available instructions.
- [ps_2_0 Registers](#) lists the different types of registers used by the vertex shader ALU.
- [Modifiers](#) Are used to modify the way an instruction works.
- [Destination Register Write Mask](#) determines what components of the destination register get written.
- [Pixel Shader Source Register Modifiers](#) alter the source register data before the instruction runs.
- [Source Register Swizzling](#) gives additional control over which register components are read, copied, or written.

Instruction Count

Shaders have restrictions for maximum instruction counts. Total Instruction slots: 96 (64 arithmetic and 32 texture).

Sampler Count

The number of texture samplers available is 16.

Related topics

[Pixel Shaders](#)

ps_2_x

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

A programmable pixel shader is made up of a set of instructions that operate on pixel data. Registers transfer data in and out of the ALU. Additional control can be applied to modify the instruction, the results, or what data gets written out.

- [ps_2_x Instructions](#) contains a list of the available instructions.
- [ps_2_x Registers](#) lists the different types of registers used by the vertex shader ALU.
- [Modifiers](#) Are used to modify the way an instruction works.
- [Destination Register Write Mask](#) determines what components of the destination register get written.
- [Pixel Shader Source Register Modifiers](#) alter the source register data before the instruction runs.
- [Source Register Swizzling](#) gives additional control over which register components are read, copied, or written.

Dynamic Flow Control

[DynamicFlowControlDepth](#) represents the nesting depth of dynamic flow control instructions: [if](#), [if_comp](#), [if_pred](#), [break - ps](#), and [break_comp - ps](#). The value is equal to the nesting depth of the if_comp block. If this cap is zero, the device does not support dynamic flow control instructions.

Number of Temporary Registers

The number of temporary registers supported by the device. The range is from 12 to 32.

Static Flow Control Nesting Depth

[StaticFlowControlDepth](#) represents the nesting depth of two types of static flow control instructions: [loop /rep](#) And [call /callnz](#). loop /rep instructions can be nested up to [StaticFlowControlDepth](#) deep. Independently, call /callnz instructions can be nested up to [StaticFlowControlDepth](#) deep.

Number of Instruction Slots

The number of instruction slots can range from 96 to a maximum of 512, and is specified by the [MaxPixelShaderInstructionSlots](#). The total number of instructions that

can run is defined by **MaxPixelShaderInstructionsExecuted**. This can be larger than the number of instruction slots due to looping and subroutine calls.

Arbitrary Swizzle

If **D3DD3DPSHADERCAPS2_0_ARBITRARYSWIZZLE** is set, arbitrary swizzle is supported. See Source Register Swizzling.

Gradient Instructions

If **D3DD3DPSHADERCAPS2_0_GRADIENTINSTRUCTIONS** is set, gradient instructions are supported. See `dsx - ps`, `dsy - ps`, and `texldd - ps`.

Predication

If **D3DD3DPSHADERCAPS2_0_PREDICATION** is set, instruction predication is supported. See Predicate Register.

Dependent Read Limit

If **D3DD3DPSHADERCAPS2_0_NODEPENDENTREADLIMIT** is set, there are no dependent read limits.

Texture Instruction Limit

If **D3DD3DPSHADERCAPS2_0_NOTEXINSTRUCTIONLIMIT** is set, there is no limit on texture instructions.

Sampler Count

The number of texture samplers available is 16.

Related topics

[Pixel Shaders](#)

ps_3_0

Article • 06/11/2021 • 3 minutes to read

A programmable pixel shader is made up of a set of instructions that operate on pixel data. Registers transfer data in and out of the ALU. Additional control can be applied to modify the instruction, the results, or what data gets written out.

- [ps_3_0 Instructions](#) contains a list of the available instructions.
- [ps_3_0 Registers](#) lists the different types of registers used by the pixel shader ALU.
- [Modifiers](#) Are used to modify the way an instruction works.
- [Destination Register Write Mask](#) determines what components of the destination register get written.
- [Pixel Shader Source Register Modifiers](#) alter the source register data before the instruction runs.
- [Source Register Swizzling](#) gives additional control over which register components are read, copied, or written.

New Features

Add a face register. Add a position register. Color registers (v#) are now fully floating point and the texture coordinate registers (t#) have been consolidated. Input declarations take the usage names, and multiple usages are permitted for components of a given register.

Dynamic Flow Control

The device supports dynamic flow control ([if bool - ps](#), [break - ps](#), and [break_comp - ps](#)). The depth of nesting ranges from 0 to 24.

Number of Temporary Registers

The number of temporary registers supported is 32.

Static Flow Control Nesting Depth

The [call - ps/callnz /call_pred](#) can be nested to a maximum depth of 4. Independently, [loop - ps/rep - ps](#) instructions can be nested to a maximum depth of 4.

Arbitrary Swizzle

Arbitrary swizzle is supported. See [Source Register Swizzling](#).

Gradient Instructions

Gradient instructions are supported. See [dsx - ps](#), [dsy - ps](#), and [texldd - ps](#).

Predication

Instruction predication is supported. See [Predicate Register](#).

Dependent Read Limit

There are no dependent read limits.

Texture Instruction Limit

There is no limit on texture instructions.

Instruction Count

Each pixel shader is allowed anywhere from 512 up to the number of slots in MaxPixelShader30InstructionSlots (not more than 32768). The number of instructions run can be much higher because of the looping support.
MaxPShaderInstructionsExecuted should be at least 2^{16} .

Sampler Count

The number of texture samplers available is 16.

Device Caps

If ps_3_0 is supported, the following caps are supported in hardware (at a minimum):

Cap	Value
MaxTextureWidth,	4K each
MaxTextureHeight	

Cap	Value
MaxTextureRepeat	8K
MaxAnisotropy	16
PixelShaderVersion	3_0
MaxPixelShader30InstructionSlots	512
The following primitive caps are set:	D3DPMISCCAPS_BLENDOP, D3DPMISCCAPS_CLIPPLANESCALEDPOLYPOINTS, D3DPMISCCAPS_CLIPTLVERTS, D3DPMISCCAPS_CULLCCW, D3DPMISCCAPS_CULLCW, D3DPMISCCAPS_CULLNONE, D3DPMISCCAPS_FOGINFVF, D3DPMISCCAPS_MASKZ
The following raster caps are set:	D3DPRASTERCAPS_MIPMAPLODBIAS, D3DPRASTERCAPS_ANISOTROPY, D3DPRASTERCAPS_COLORPERSPECTIVE, D3DPRASTERCAPS_SCISSORTEST in D3DCAPS9
Full support for depth bias including:	D3DPRASTERCAPS_SLOPESCALEDEPTHBIAS, D3DPRASTERCAPS_DEPTHBIAS
Full set of comparisons for depth and alpha test including:	All the D3DPCMPcaps in D3DCAPS9.
Source blending modes	All blending modes are supported as a source (except D3DPBLENDCAPS_SRCALPHASAT, D3DPBLENDCAPS_BOTHSRCALPHA, and D3DPBLENDCAPS_BOTHINVSRCALPHA).
The following texture caps are supported:	D3DPTEXTURECAPS_CUBEMAP, D3DPTEXTURECAPS_MIPCUBEMAP, D3DPTTEXTURECAPS_MIPMAP, D3DPTTEXTURECAPS_MIPVOLUMEMAP, D3DPTTEXTURECAPS_PERSPECTIVE, D3DPTTEXTURECAPS_PROJECTED, D3DPTTEXTURECAPS_TEXREPEATNOTSCALEDBYSIZE, D3DPTTEXTURECAPS_VOLUMEMAP
The following are supported on texture filter caps, volume texture filter caps and cube texture filter caps:	D3DPTFILTERCAPS_MINPOINT, D3DPTFILTERCAPS_MINLINEAR, D3DPTFILTERCAPS_MINFANISOTROPIC (This is not required for VolumeTextureFilterCaps and CubeTextureFilterCaps), D3DPTFILTERCAPS_MIPFPOINT, D3DPTFILTERCAPS_MIPFLINEAR, D3DPTFILTERCAPS_MAGPOINT, D3DPTFILTERCAPS_MAGLINEAR

Cap	Value
The following texture address modes are supported at vertex and pixel stages:	D3DPTADDRESSCAPS_WRAP, D3DPTADDRESSCAPS_MIRROR, D3DPTADDRESSCAPS_CLAMP, D3DPTADDRESSCAPS_BORDER, D3DPTADDRESSCAPS_INDEPENDENTUV, D3DPTADDRESSCAPS_MIRRORONCE
All the pixel shader caps are supported.	DynamicFlowControlDepth = 24, NumTemps = 32, StaticFlowControlDepth = 4, NumInstructionSlots = 512. The following features are supported: predication, arbitrary swizzles, and gradient instructions. There is no dependent-read limit, and no limit on the mixture of texture and math instructions.
All the stencil operations are supported. This includes two sided stencil.	See D3DSTENCILOP
Device support point size per vertex	D3DFVFCAPS_PSIZE in D3DCAPS9
Non-power of 2 texture support.	Either full support or conditional non-pow-2 support; device should not have the square texture only limitation as in D3DPTEXTURECAPS_SQUAREONLY.
If the device supports multiple rendertargets, the following caps are supported:	D3DPMISCCAPS_INDEPENDENTWRITEMASKS, D3DPMISCCAPS_MRTPOSTPIXELSHADERBLENDING
If vs_3_0 is supported	MaxUserClipPlanes in D3DCAPS9 is 6

Related topics

[Pixel Shaders](#)

Pixel Shader Differences

Article • 08/19/2020 • 2 minutes to read

Instruction Slots

Each version supports a different number of maximum instruction slots.

Version	Maximum number of instruction slots
ps_1_1	4 texture + 8 arithmetic
ps_1_2	4 texture + 8 arithmetic
ps_1_3	4 texture + 8 arithmetic
ps_1_4	6 texture + 8 arithmetic per phase
ps_2_0	32 texture + 64 arithmetic
ps_2_x	96 minimum, and up to the number of slots in D3DCAPS9.D3DPSHADERCAPS2_0.NumInstructionSlots. See D3DPSHADERCAPS2_0.
ps_3_0	512 minimum, and up to the number of slots in D3DCAPS9.MaxPixelShader30InstructionSlots. See D3DPSHADERCAPS2_0.

For information about the limitations of software shaders, see [Software Shaders](#).

Flow Control Nesting Limits

- See [Flow Control Limitations](#).

ps_1_x Features

New instructions:

See [ps_1_1, ps_1_2, ps_1_3, ps_1_4 Instructions](#).

New registers:

See [ps_1_1_ps_1_2_ps_1_3_ps_1_4 Registers](#).

ps_2_0 Features

New features:

- Three new swizzles - .yzxw, .zxyw, .wzyx
- Number of **Temporary Register** (r#) increased to 12
- Number of **Constant Float Register** registers (c#) increased to 32
- Number of **Texture Coordinate Registers** (t#) increased to 8

New instructions:

- Setup instructions - `dcl` - (sm2, sm3 - ps asm), `dcl_samplerType` (sm2, sm3 - ps asm)
- Arithmetic instructions - `abs` - ps, `crs` - ps, `dp2add` - ps, `exp` - ps, `frc` - ps, `log` - ps, `m3x2` - ps, `m3x3` - ps, `m3x4` - ps, `m4x3` - ps, `m4x4` - ps, `max` - ps, `min` - ps, `nrm` - ps, `pow` - ps, `rcp` - ps, `rsq` - ps, `sincos` - ps
- Texture instructions - `texld` - ps_2_0 and up (different syntax), `texldb` - ps, `texldp` - ps

New registers:

- Sampler (Direct3D 9 asm-ps) (s#)

ps_2_x Features

New features (See [D3DPSHADERCAPS2_0](#)):

- Dynamic flow control
- Static flow control
- Nesting for dynamic and static flow control instructions
- Number of **Temporary Registers** (r#) increased
- Arbitrary source swizzle
- Gradient instructions
- Predication
- No dependent texture read limit
- No texture instruction limit

New instructions:

- Static flow control instructions - `if` bool - ps, `call` - ps, `callnz` bool - ps, `else` - ps, `endif` - ps, `rep` - ps, `endrep` - ps, `label` - ps, `ret` - ps
- Dynamic flow control instructions - `break` - ps, `break_comp` - ps, `breakp` - ps, `callnz` pred - ps, `if_comp` - ps, `if_pred` - ps, `setp_comp` - ps

- Arithmetic instructions - [dsx - ps](#), [dsy - ps](#)
- Texture instruction - [texldd - ps](#)

New registers:

- [Predicate Register](#) (p0)

ps_3_0 Features

New features:

- Consolidated 10 [Input Registers](#) (v#)
- Indexable [Input Color Register](#) (v#) with the [Loop Counter Register](#) (aL)
- Number of [Temporary Registers](#) (r#) increased to 32
- Number of [Constant Float Registers](#) (c#) increased to 224

New instructions:

- Setup instruction - [dcl_semantics](#) (sm3 - ps asm)
- Static flow instructions - [loop - ps](#), [endloop - ps](#)
- Arithmetic instruction - [sincos - ps](#) (new syntax)
- Texture instruction - [texldl - ps](#)

New registers:

- [Input Register](#) (v#)
- [Position Register](#) (vPos)
- [Face Register](#) (vFace)

Related topics

[Pixel Shaders](#)

Pixel Shader Instructions

Article • 08/23/2019 • 2 minutes to read

- [ps_1_1, ps_1_2, ps_1_3, ps_1_4 Instructions](#)
- [ps_2_0 Instructions](#)
- [ps_2_x Instructions](#)
- [ps_3_0 Instructions](#)
- [Modifiers for ps_1_X](#)
- [Flow Control Limitations](#)

Related topics

[Pixel Shaders](#)

ps_1_1, ps_1_2, ps_1_3, ps_1_4 Instructions

Article • 06/30/2021 • 2 minutes to read

This section contains reference information for the pixel shader version 1_X instructions.

There are several types of pixel shader instructions, as shown in the following table.

Instruction Set

Version	Description	Instruction slots	1_1	1_2	1_3	1_4
ps	Version number	0	x	x	x	x
Constant instructions			1_1	1_2	1_3	1_4
def - ps	Define constants	0	x	x	x	x
Phase instructions			1_1	1_2	1_3	1_4
phase - ps	Transition between phase 1 and phase 2	0				x
Arithmetic instructions			1_1	1_2	1_3	1_4
add - ps	Add two vectors	1	x	x	x	x
bem - ps	Apply a fake bump environment-map transform	2				x
cmp - ps	Compare source to 0	1 ¹	x	x	x	x
cnd - ps	Compare source to 0.5	1	x	x	x	x
dp3 - ps	Three-component dot product	1	x	x	x	x
dp4 - ps	Four-component dot product	1 ¹	x	x	x	x
lrp - ps	Linear interpolate	1	x	x	x	x
mad - ps	Multiply and add	1	x	x	x	x
mov - ps	Move	1	x	x	x	x
mul - ps	Multiply	1	x	x	x	x
nop - ps	No operation	0	x	x	x	x

Version	Description	Instruction slots	1_1	1_2	1_3	1_4
sub - ps	Subtract	1	x	x	x	x
Texture instructions			1_1	1_2	1_3	1_4
tex - ps	Sample a texture	1	x	x	x	
texbem - ps	Apply a fake bump environment-map transform	1	x	x	x	
texbeml - ps	Apply a fake bump environment-map transform with luminance correction	1+1 ²	x	x	x	
texcoord - ps	Interpret texture coordinate data as color data	1	x	x	x	
texcrd - ps	Copy texture coordinate data as color data	1				x
texdepth - ps	Calculate depth values	1				x
texdp3 - ps	Three-component dot product between texture data and the texture coordinates	1		x	x	
texdp3tex - ps	Three-component dot product and 1D texture lookup	1		x	x	
texkill - ps	Cancels rendering of pixels based on a comparison	1	x	x	x	x
texld - ps_1_4	Sample a texture	1				x
texm3x2depth - ps	Calculate per-pixel depth values	1				x
texm3x2pad - ps	First row matrix multiply of a two-row matrix multiply	1	x	x	x	
texm3x2tex - ps	Final row matrix multiply of a two-row matrix multiply	1	x	x	x	
texm3x3 - ps	3x3 matrix multiply	1		x	x	
texm3x3pad - ps	First or second row multiply of a three-row matrix multiply	1	x	x	x	
texm3x3spec - ps	Final row multiply of a three-row matrix multiply	1	x	x	x	

Version	Description	Instruction slots	1_1	1_2	1_3	1_4
texm3x3tex - ps	Texture look up using a 3x3 matrix multiply	1	x	x	x	
texm3x3vspec - ps	Texture look up using a 3x3 matrix multiply, with non-constant eye-ray vector	1	x	x	x	
texreg2ar - ps	Sample a texture using the alpha and red components	1	x	x	x	
texreg2gb - ps	Sample a texture using the green and blue components	1	x	x	x	
texreg2rgb - ps	Sample a texture using the red, green and blue components	1		x	x	

1. 1 slot in ps_1_4; 2 slots in ps_1_2 and ps_1_3

2. $1 + 1 = 1$ arithmetic instruction + 1 texture instruction

Related topics

[Pixel Shader Instructions](#)

ps_2_0 Instructions

Article • 08/23/2019 • 2 minutes to read

This section contains reference information for the pixel shader version 2_0 instructions.

There are several types of pixel shader instructions, as shown in the table. Columns to the right mean the following:

- Instruction slots - Number of instruction slots used by each instruction.
- Setup - A pixel shader must have a version instruction and it must be the first instruction.
- Arithmetic - These instructions provide the mathematical operations in a shader.
- Texture - These instructions are used to load and sample texture data, and to modify texture coordinates.
- New - These instructions are new to this version.

Instruction Set

Name	Description	Instruction slots	Setup	Arithmetic	Texture	New
abs - ps	Absolute value	1		x		x
add - ps	Add two vectors	1		x		
cmp - ps	Compare source to 0	1		x		
crs - ps	Cross product	2		x		x
dcl_samplerType (sm2, sm3 - ps asm)	Declare the texture dimension for a sampler	0	x			x
dcl - (sm2, sm3 - ps asm)	Declare the association between vertex shader output registers and pixel shader input registers.	0	x			x
def - ps	Define constants	0	x			
dp2add - ps	2D dot product and add	2		x		x
dp3 - ps	3D dot product	1		x		

Name	Description	Instruction slots	Setup	Arithmetic	Texture	New
dp4 - ps	4D dot product	1		x		
exp - ps	Full precision 2^x	1		x		x
frc - ps	Fractional component	1		x		x
log - ps	Full precision $\log_2(x)$	1		x		x
lrp - ps	Linear interpolate	2		x		
m3x2 - ps	3x2 multiply	2		x		x
m3x3 - ps	3x3 multiply	3		x		x
m3x4 - ps	3x4 multiply	4		x		x
m4x3 - ps	4x3 multiply	3		x		x
m4x4 - ps	4x4 multiply	4		x		x
mad - ps	Multiply and add	1		x		
max - ps	Maximum	1		x		x
min - ps	Minimum	1		x		x
mov - ps	Move	1		x		
mul - ps	Multiply	1		x		
nop - ps	No operation	1		x		
nrm - ps	Normalize	3		x		x
pow - ps	x^y	3		x		x
ps	Version	0	x			
rcp - ps	Reciprocal	1		x		x
rsq - ps	Reciprocal square root	1		x		x
sincos - ps	Sine and cosine	8		x		x
sub - ps	Subtract	1		x		
texkill - ps	Kill pixel render	1			x	
texld - ps_2_0 and up	Sample a texture	1		x	x	

Name	Description	Instruction slots	Setup	Arithmetic	Texture	New
texldb - ps	Texture sampling with level-of-detail bias from w-component	1			x	x
texldp - ps	Texture sampling with projective divide by w-component	1			x	x

Related topics

[Pixel Shader Instructions](#)

ps_2_x Instructions

Article • 08/19/2020 • 3 minutes to read

This section contains reference information for the pixel shader version 2_x instructions.

There are several types of pixel shader instructions, as shown in the table. Columns to the right mean the following:

- Instruction slots - Number of instruction slots used by each instruction.
- Setup - A pixel shader must have a version instruction and it must be the first instruction.
- Arithmetic - These instructions provide the mathematical operations in a shader.
- Texture - These instructions are used to load and sample texture data, and to modify texture coordinates.
- Flow control - These instructions provide static and dynamic flow control to the execution of instructions.
- New - These instructions are new to this version.

Instruction Set

Name	Description	Instruction slots	Setup	Arithmetic	Texture	Flow control	New control
abs - ps	Absolute value	1		x			
add - ps	Add two vectors	1		x			
break - ps	Break out of a rep...endrep block	1			x	x	
break_comp - ps	Conditionally break out of a rep...endrep block, with a comparison	3			x	x	

Name	Description	Instruction slots	Setup	Arithmetic	Texture	Flow	New control
breakp - ps	Break out of a rep...endrep block, based on a predicate	3				x	x
call - ps	Call a subroutine	2				x	x
callnz bool - ps	Call a subroutine if a boolean register is not zero	3				x	x
callnz pred - ps	Call a subroutine if a predicate register is not zero	3				x	x
cmp - ps	Compare source to 0	1			x		
crs - ps	Cross product	2			x		
dcl_samplerType (sm2, sm3 - ps asm)	Declare the texture dimension for a sampler	0		x			
dcl - (sm2, sm3 - ps asm)	Declare the association between vertex shader output registers and pixel shader input registers.	0		x			
def - ps	Define constants	0		x			

Name	Description	Instruction slots	Setup	Arithmetic	Texture	Flow	New control
defb - ps	Define a Boolean constant	0	x			x	
defi - ps	Define an integer constant	0	x			x	
dp2add - ps	2D dot product and add	2		x			
dp3 - ps	3D dot product	1		x			
dp4 - ps	4D dot product	1		x			
dsx - ps	Rate of change in the x-direction	2		x		x	
dsy - ps	Rate of change in the y direction	2		x		x	
else - ps	Begin an else block	1			x	x	
endif - ps	End an if...else block	1			x	x	
endrep - ps	End of a repeat block	2			x	x	
exp - ps	Full precision 2^x	1		x			
frc - ps	Fractional component	1		x			
if bool - ps	Begin an if block	3			x	x	

Name	Description	Instruction slots	Setup	Arithmetic	Texture	Flow	New control
if_comp - ps	Begin an if block with a comparison	3				x	x
if pred - ps	Begin an if block with predication	3				x	x
label - ps	Label	0				x	x
log - ps	Full precision $\log_2(x)$	1			x		
lrc - ps	Linear interpolate	2			x		
m3x2 - ps	3x2 multiply	2			x		
m3x3 - ps	3x3 multiply	3			x		
m3x4 - ps	3x4 multiply	4			x		
m4x3 - ps	4x3 multiply	3			x		
m4x4 - ps	4x4 multiply	4			x		
mad - ps	Multiply and add	1			x		
max - ps	Maximum	1			x		
min - ps	Minimum	1			x		
mov - ps	Move	1			x		
mul - ps	Multiply	1			x		
nop - ps	No operation	1			x		
nrm - ps	Normalize	3			x		
pow - ps	x^y	3			x		
ps	Version	0	x				
rcp - ps	Reciprocal	1			x		
rep - ps	Repeat	3				x	x

Name	Description	Instruction slots	Setup	Arithmetic	Texture	Flow	New control
ret - ps	End of a subroutine	1				x	x
rsq - ps	Reciprocal square root	1		x			
setp_comp	Set the predicate register	1				x	x
sincos - ps	Sine and cosine	8		x			
sub - ps	Subtract	1		x			
texkill - ps	Kill pixel render		See note 1		x		
texld - ps_2_0 and up	Sample a texture		See note 2		x		
texldb - ps	Texture sampling with level-of-detail bias from w-component		See note 3		x		
texldd - ps	Texture sampling with user-provided gradients	3			x		x
texldp - ps	Texture sampling with projective divide by w-component		See note 4		x		

Notes:

1. If **D3DD3DP王某CAPS2_0_NOTEINSTRUCTIONLIMIT** is set, slots = 2; otherwise slots = 1.

2. If **D3DD3DP王某2_0_NOTEINSTRUCTIONLIMIT** is set and the texture is a cube map, slots = 4; otherwise slot = 1.
3. If **D3DD3DP王某2_0_NOTEINSTRUCTIONLIMIT** is set, slots = 6; otherwise slots = 1.
4. If **D3DD3DP王某2_0_NOTEINSTRUCTIONLIMIT** is not set, slots = 1; otherwise:
 - if **D3DD3DP王某2_0_NOTEINSTRUCTIONLIMIT** is set and the texture is a cube map, slots = 4.
 - if **D3DD3DP王某2_0_NOTEINSTRUCTIONLIMIT** is set and the texture is not a cube map, slots = 3.

Related topics

[Pixel Shader Instructions](#)

ps_3_0 Instructions

Article • 08/23/2019 • 3 minutes to read

This section contains reference information for the pixel shader version 3_0 instructions.

There are several types of pixel shader instructions, as shown in the table. Columns to the right mean the following:

- Instruction slots - Number of instruction slots used by each instruction.
- Setup - A pixel shader must have a version instruction and it must be the first instruction.
- Arithmetic - These instructions provide the mathematical operations in a shader.
- Texture - These instructions are used to load and sample texture data, and to modify texture coordinates.
- Flow control - These instructions provide static and dynamic flow control to the execution of instructions.
- New - These instructions are new to this version.

Instruction Set

Name	Description	Instruction slots	Setup	Arithmetic	Texture	Flow control	New control
abs - ps	Absolute value	1		x			
add - ps	Add two vectors	1		x			
break - ps	Break out of a loop...endloop or rep...endrep block	1				x	
break_comp - ps	Conditionally break out of a loop...endloop or rep...endrep block, with a comparison	3				x	

Name	Description	Instruction slots	Setup	Arithmetic	Texture	Flow	New control
breakp - ps	break out of a loop...endloop or rep...endrep block, based on a predicate	3				x	
call - ps	Call a subroutine	2				x	
callnz bool - ps	Call a subroutine if a boolean register is not zero	3				x	
callnz pred - ps	Call a subroutine if a predicate register is not zero	3				x	
cmp - ps	Compare source to 0	1			x		
crs - ps	Cross product	2			x		
dcl_samplerType (sm2, sm3 - ps asm)	Declare the texture dimension for a sampler	0		x			
dcl_semantics (sm3 - ps asm)	Declare input and output registers	0		x			x
def - ps	Define constants	0		x			
defb - ps	Define a Boolean constant	0		x			
defi - ps	Define an integer constant	0		x			

Name	Description	Instruction slots	Setup	Arithmetic	Texture	Flow	New control
dp2add - ps	2D dot product and add	2		x			
dp3 - ps	3D dot product	1		x			
dp4 - ps	4D dot product	1		x			
dsx - ps	Rate of change in the x-direction	2		x			
dsy - ps	Rate of change in the y direction	2		x			
else - ps	Begin an else block	1				x	
endif - ps	End an if...else block	1				x	
endloop - ps	End a loop	2				x	x
endrep - ps	End of a repeat block	2				x	
exp - ps	Full precision 2^x	1		x			
frc - ps	Fractional component	1		x			
if bool - ps	Begin an if block	3				x	
if_comp - ps	Begin an if block with a comparison	3				x	
if pred - ps	Begin an if block with predication	3				x	
label - ps	Label	0				x	

Name	Description	Instruction slots	Setup	Arithmetic	Texture	Flow	New control
log - ps	Full precision log ₂ (x)	1		x			
loop - ps	Loop	3				x	x
lrp - ps	Linear interpolate	2		x			
m3x2 - ps	3x2 multiply	2		x			
m3x3 - ps	3x3 multiply	3		x			
m3x4 - ps	3x4 multiply	4		x			
m4x3 - ps	4x3 multiply	3		x			
m4x4 - ps	4x4 multiply	4		x			
mad - ps	Multiply and add	1		x			
max - ps	Maximum	1		x			
min - ps	Minimum	1		x			
mov - ps	Move	1		x			
mul - ps	Multiply	1		x			
nop - ps	No operation	1		x			
nrm - ps	Normalize	3		x			
pow - ps	x^y	3		x			
ps	Version	0	x				
rcp - ps	Reciprocal	1		x			
rep - ps	Repeat	3				x	
ret - ps	End of a subroutine	1				x	
rsq - ps	Reciprocal square root	1		x			
setp_comp	Set the predicate register	1				x	

Name	Description	Instruction slots	Setup	Arithmetic	Texture	Flow	New control
sincos - ps	Sine and cosine	8		x			
sub - ps	Subtract	1		x			
texkill - ps	Kill pixel render	2			x		
texld - ps_2_0 and up	Sample a texture	See note 1			x		
texldb - ps	Texture sampling with level-of-detail bias from w-component	6			x		
texldl - ps	Texture sampling with level-of-detail from w-component	See note 2			x		x
texldd - ps	Texture sampling with user-provided gradients	3			x		
texldp - ps	Texture sampling with projective divide by w-component	See note 3			x		

Notes:

1. If the texture is a cube map, slots = 4; otherwise slots = 1.
2. If the texture is a cube map, slots = 5; otherwise slots = 2.
3. If the texture is a cube map, slots = 4; otherwise slots = 3.

Related topics

[Pixel Shader Instructions](#)

Modifiers for ps_1_X

Article • 06/30/2021 • 2 minutes to read

Instruction modifiers affect the result of the instruction before it is written into the destination register. For instance, use them to multiply or divide the result by a factor of two, or to clamp the result between zero and one. Instruction modifiers are applied after the instruction runs but before writing the result to the destination register.

A list of the modifiers is shown below.

Modifier	Description	Syntax	Version	Version	Version	Version
			1_1	1_2	1_3	1_4
_x2	Multiply by 2	instruction_x2	X	X	X	X
_x4	Multiply by 4	instruction_x4	X	X	X	X
_x8	Multiply by 8	instruction_x8				X
_d2	Divide by 2	instruction_d2	X	X	X	X
_d4	Divide by 4	instruction_d4				X
_d8	Divide by 8	instruction_d8				X
_sat	Saturate (clamp from 0 and 1)	instruction_sat	X	X	X	X

- The multiply modifier multiplies the register data by a power of two after it is read. This is the same as a shift left.
- The divide modifier divides the register data by a power of two after it is read. This is the same as a shift right.
- The saturate modifier clamps the range of register values from zero to one.

Instruction modifiers can be used on arithmetic instructions. They may not be used on texture address instructions.

Multiply modifier

This example loads the destination register (dest) with the sum of the two colors in the source operands (src0 and src1) and multiplies the result by two.

```
add_x2 dest, src0, src1
```

This example combines two instruction modifiers. First, two colors in the source operands (src0 and src1) are added. The result is then multiplied by two, and clamped between 0.0 to 1.0 for each component. The result is saved in the destination register.

```
add_x2_sat dest, src0, src1
```

Divide modifier

This example loads the destination register (dest) with the sum of the two colors in the source operands (src0 and src1) and divides the result by two.

```
add_d2 dest, src0, src1
```

Saturate modifier

For arithmetic instructions, the saturation modifier clamps the result of this instruction into the range 0.0 to 1.0 for each component. The following example shows how to use this instruction modifier.

```
dp3_sat r0, t0_bx2, v0_bx2 ; t0 is bump, v0 is light direction
```

This operation occurs after any multiply or divide instruction modifier. _sat is most often used to clamp dot product results. However, it also enables consistent emulation of multipass methods where the frame buffer is always in the range 0 to 1, and of DirectX 6 and 7.0 multitexture syntax, in which saturation is defined to occur at every stage.

This example loads the destination register (dest) with the sum of the two colors in the source operands (src0 and src1), and clamps the result into the range 0.0 to 1.0 for each component.

```
add_sat dest, src0, src1
```

This example combines two instruction modifiers. First, two colors in the source operands (`src0` and `src1`) are added. The result is multiplied by two and clamped between 0.0 to 1.0 for each component. The result is saved in the destination register.

```
add_x2_sat dest, src0, src1
```

Related topics

[Pixel Shader Instructions](#)

Modifiers for ps_2_0 and Above

Article • 06/10/2021 • 2 minutes to read

Instruction modifiers affect the result of the instruction before it is written into the destination register.

This section contains reference information for the instruction modifiers implemented by pixel shader version 2_0 and above.

Name	Syntax	2_0	2_x	2_sw	3_0	3_sw
Centroid	_centroid	x	x	x	x	x
Partial_Precision	_pp	x	x	x	x	x
Saturate	_sat	x	x	x	x	x

Centroid

The centroid modifier is an optional modifier that clamps attribute interpolation within the range of the primitive when a multisample pixel center is not covered by the primitive. This can be seen in [Centroid Sampling ↗](#).

You can apply the centroid modifier to an assembly instruction as shown here:

```
dcl_texcoord0_centroid v0
```

You can also apply the centroid modifier to a semantic as shown here:

```
float4 TexturePointCentroidPS( float4 TexCoord : TEXCOORD0_centroid ) :  
COLOR0  
{  
    return tex2D( PointSampler, TexCoord );  
}
```

In addition, any [Input Color Register](#) (v#) declared with a color semantic will automatically have centroid applied. Gradients computed from attributes that are centroid sampled are not guaranteed to be accurate.

Partial Precision

The partial precision instruction modifier (_pp) indicates areas where partial precision is acceptable, provided that the underlying implementation supports it. Implementations are always free to ignore the modifier and perform the affected operations in full precision.

The _pp modifier can occur in two contexts:

- On a texture coordinate declaration to enable passing texture coordinates to the pixel shader in partial-precision form. This allows, for example, the use of texture coordinates to relay color data to the pixel shader, which may be faster with partial precision than with full precision in some implementations. In the absence of this modifier, texture coordinates must be passed in full precision.
- On any instruction including texture load instructions. This indicates that the implementation is allowed to execute the instruction with partial precision and store a partial precision result. In the absence of an explicit modifier, the instruction must be performed at full precision (regardless of the input precision).

Examples:

```
dcl_texcoord0_pp t1  
cmp_pp r0, r1, r2, r3
```

Saturate

The saturate instruction modifier (_sat) saturates (or clamps) the instruction result to the range [0, 1] before writing to the destination register.

The _sat instruction modifier can be used with any instruction except [frc - ps](#), [sincos - ps](#), and any tex* instructions.

For ps_2_0, ps_2_x, and ps_2_sw, the _sat instruction modifier cannot be used with instructions writing to any output registers ([Output Color Register](#) or [Output Depth Register](#)). This restriction does not apply to ps_3_0 and above.

Example:

```
dp3_sat r0, v0, c1
```

Related topics

[Pixel Shader Instructions](#)

Flow Control Limitations

Article • 08/23/2019 • 8 minutes to read

Pixel shader flow control instructions have limits affecting how many levels of nesting can be included in the instructions. In addition, there are some limitations for implementing per-pixel flow control with gradient instructions.

ⓘ Note

When you use the *_4_0_level_9_x HLSL shader profiles, you implicitly use of the **Shader Model 2.x** profiles to support Direct3D 9 capable hardware. Shader Model 2.x profiles support more limited flow control behavior than the **Shader Model 4.x** and later profiles.

- [Pixel Shader Instruction Depth Counts](#)
- [Interaction of Per-Pixel Flow Control With Screen Gradients](#)

Pixel Shader Instruction Depth Counts

ps_2_0 does not support flow control. The limitations for the other pixel shader versions are listed below.

Instruction Depth Count for ps_2_x

Each instruction counts against one or more nesting depth limits. The following table lists the depth count that each instruction adds or subtracts from the existing depth.

Instruction	Static nesting	Dynamic nesting	loop/rep nesting	call nesting
if bool - ps	1	0	0	0
if_comp - ps	0	1	0	0
if pred - ps	0	1	0	0
else - ps	0	0	0	0
endif - ps	-1(if bool - ps)	-1(if pred - ps or if_comp - ps)	0	0

Instruction	Static nesting	Dynamic nesting	loop/rep nesting	call nesting
rep - ps	0	0	1	0
endrep - ps	0	0	-1	0
break - ps	0	0	0	0
break_comp - ps	0	1, -1	0	0
breakp - ps	0	0	0	0
call - ps	0	0	0	1
callnz bool - ps	0	0	0	1
callnz pred - ps	0	1	0	1
ret - ps	0	-1(callnz pred - ps)	0	-1
setp_comp - ps	0	0	0	0

Nesting Depth

Nesting depth defines the number of instructions can be called from inside of each other. Each type of instruction has one or more nesting limits as indicated in the following table.

Instruction Type	Maximum
Static nesting	24 if (D3DCAPS9.D3DPSHADERCAPS2_0.StaticFlowControlDepth > 0); 0 otherwise
Dynamic nesting	0 to 24, see D3DCAPS9.D3DPSHADERCAPS2_0.DynamicFlowControlDepth
rep nesting	0 to 4, see D3DCAPS9.D3DPSHADERCAPS2_0.StaticFlowControlDepth
call nesting	0 to 4, see D3DCAPS9.D3DPSHADERCAPS2_0.StaticFlowControlDepth (independent of rep limit)

Instruction Depth Count for ps_2_sw

Each instruction counts against one or more nesting depth limits. This table shows the depth count that each instruction adds or subtracts from the existing depth.

Instruction	Static nesting	Dynamic nesting	loop/rep nesting	call nesting
if bool - ps	1	0	0	0
if pred - ps	0	1	0	0
if_comp - ps	0	1	0	0
else - ps	0	0	0	0
endif - ps	-1(if bool - ps)	-1(if pred - ps or if_comp - ps)	0	0
rep - ps	0	0	1	0
endrep - ps	0	0	-1	0
loop - ps	n/a	n/a	n/a	n/a
endloop - ps	n/a	n/a	n/a	n/a
break - ps	0	0	0	0
break_comp - ps	0	1, -1	0	0
breakp - ps	0	0	0	0
call - ps	0	0	0	1
callnz bool - ps	0	0	0	1
callnz pred - ps	0	1	0	1
ret - ps	0	-1(callnz pred - ps)	0	-1
setp_comp - ps	0	0	0	0

Nesting Depth

Nesting depth defines the number of instructions that can be called from inside of each other. Each type of instruction has one or more nesting limits as indicated in the

following table.

Instruction Type	Maximum
Static nesting	24
Dynamic nesting	24
rep nesting	4
call nesting	4

Instruction Depth Count for ps_3_0

Each instruction counts against one or more nesting depth limits. This table shows the depth count that each instruction adds or subtracts from the existing depth.

Instruction	Static nesting	Dynamic nesting	loop/rep nesting	call nesting
if bool - ps	1	0	0	0
if pred - ps	0	1	0	0
if_comp - ps	0	1	0	0
else - ps	0	0	0	0
endif - ps	-1(if bool - ps)	-1(if pred - ps or if_comp - ps)	0	0
rep - ps	0	0	1	0
endrep - ps	0	0	-1	0
loop - ps	0	0	1	0
endloop - ps	0	0	-1	0
break - ps	0	0	0	0
break_comp - ps	0	1, -1	0	0
breakp - ps	0	0	0	0
call - ps	0	0	0	1

Instruction	Static nesting	Dynamic nesting	loop/rep nesting	call nesting
callnz bool - ps	0	0	0	1
callnz pred - ps	0	1	0	1
ret - ps	0	-1(callnz pred - ps)	0	-1
setp_comp - ps	0	0	0	0

Nesting Depth

Nesting depth defines the number of instructions that can be called from inside of each other. Each type of instruction has one or more nesting limits as indicated in the following table.

Instruction Type	Maximum
Static nesting	24
Dynamic nesting	24
loop/rep nesting	4
call nesting	4

Instruction Depth Count for ps_3_sw

Each instruction counts against one or more nesting depth limits. This table shows the depth count that each instruction adds or subtracts from the existing depth.

Instruction	Static nesting	Dynamic nesting	loop/rep nesting	call nesting
if bool - ps	1	0	0	0
if pred - ps	0	1	0	0
if_comp - ps	0	1	0	0
else - ps	0	0	0	0

Instruction	Static nesting	Dynamic nesting	loop/rep nesting	call nesting
endif - ps	-1(if bool - ps)	-1(if pred - ps or if_comp - ps)	0	0
rep - ps	0	0	1	0
endrep - ps	0	0	-1	0
loop - ps	0	0	1	0
endloop - ps	0	0	-1	0
break - ps	0	0	0	0
break_comp - ps	0	1, -1	0	0
breakp - ps	0	0	0	0
call - ps	0	0	0	1
callnz bool - ps	0	0	0	1
callnz pred - ps	0	1	0	1
ret - ps	0	-1(callnz pred - ps)	0	-1
setp_comp - ps	0	0	0	0

Nesting Depth

Nesting depth defines the number of instructions that can be called from inside of each other. Each type of instruction has one or more nesting limits as indicated in the following table.

Instruction Type	Maximum
Static nesting	24
Dynamic nesting	24
loop/rep nesting	4
call nesting	4

Interaction of Per-Pixel Flow Control With Screen Gradients

The pixel shader instruction set includes several instructions that produce or use gradients of quantities with respect to screen space x and y. The most common use for gradients is to compute level-of-detail calculations for texture sampling, and in the case of anisotropic filtering, selecting samples along the axis of anisotropy. Typically, hardware implementations run the pixel shader on multiple pixels simultaneously (such as a 2x2 grid), so that gradients of quantities computed in the shader can be reasonably approximated as deltas of the values at the same point of execution in adjacent pixels.

When flow control is present in a shader, the result of a gradient calculation requested inside a given branch path is ambiguous when adjacent pixels may execute separate flow control paths. Therefore, it is deemed illegal to use any pixel shader operation that requests a gradient calculation to occur at a location that is inside a flow control construct which could vary across pixels for a given primitive being rasterized.

All pixel shader instructions are partitioned into those operations that are permitted and into those that are not permitted inside of flow control:

- Scenario A: Operations that are not permitted inside flow control that could vary across the pixels in a primitive. These include the operations listed in the following table.

Instruction	Is Permitted in Flow Control when:
<code>texld - ps_2_0 and up</code> , <code>texldb - ps</code> and <code>texldp - ps</code>	A temporary register is used for the texture coordinate.
<code>dsx - ps</code> and <code>dsy - ps</code>	A temporary register is used for the operand.

- Scenario B: Operations that are permitted anywhere. These include the operations listed in the following table.

Instruction	Is Permitted Anywhere when:
<code>texld - ps_2_0 and up</code> , <code>texldb - ps</code> and <code>texldp - ps</code>	A read-only quantity is used for the texture coordinate (may vary per-pixel, such as interpolated texture coordinates).
<code>dsx - ps</code> and <code>dsy - ps</code>	A read-only quantity is used for the input operand (may vary per-pixel, such as interpolated texture coordinates).

Instruction	Is Permitted Anywhere when:
<code>texldl - ps</code>	The user provides level-of-detail as an argument, so there are no gradients, and thus no issue with flow control.
<code>texldd - ps</code>	The user provides gradients as input arguments, so there is no issue with flow control.

These restrictions are strictly enforced in shader validation. Scenarios having a branch condition that looks like it would branch consistently across a primitive, even though an operand in the condition expression is a pixel-shader-computed quantity, nevertheless still fall into scenario A and are not permitted. Similarly, scenarios where gradients are requested on some shader-computed quantity x from inside dynamic flow control, yet where it appears that x is not modified across any of the branches, nevertheless still fall into scenario A and are not permitted.

Predication is included in these restrictions on flow control, so that implementations remain free to trivially interchange the implementation of branch instructions with predicated instructions.

The user can use instructions from scenarios A and B together. For example, suppose the user needs an anisotropic texture sample given a shader computed texture coordinate; however, the texture load is only needed for pixels satisfying some per-pixel condition. To meet these requirements, the user can compute the texture coordinate for all pixels, outside per-pixel varying flow control, immediately computing gradients using `dsx - ps` and `dsy - ps` instructions. Then, within a per-pixel `if bool - ps/endif - ps` block, the user can use `texldd - ps` (texture load with user provided gradients), passing the precalculated gradients. Another way to describe this usage pattern is that, while all pixels in the primitive had to compute the texture coordinates and be involved with gradient calculation, only the pixels that needed to sample a texture actually did so.

Regardless of these rules, the burden is still on the user to ensure that before computing any gradient (or performing a texture sample that implicitly computes a gradient), the register containing the source data must have been initialized for all execution paths beforehand. Initialization of temporary registers is not validated or enforced in general.

Related topics

[Pixel Shader Instructions](#)

abs - ps

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

Computes absolute value.

Syntax

`abs dst, src`

where

- dst is the destination register.
- src is a source register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
abs					x	x	x	x	x

This instruction works as shown here.

```
dest.x = abs(src.x)
dest.y = abs(src.y)
dest.z = abs(src.z)
dest.w = abs(src.w)
```

Instruction Information

Requirement	Value
Minimum operating system	Windows 98

Related topics

Pixel Shader Instructions

add - ps

Article • 06/30/2021 • 2 minutes to read

Adds two vectors.

Syntax

```
add dst, src0, src1
```

where

- dst is the destination register.
- src0 is a source register.
- src1 is a source register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
add	x	x	x	x	x	x	x	x	x

The following code snippet shows the operations performed.

```
dest.x = src0.x + src1.x;  
dest.y = src0.y + src1.y;  
dest.z = src0.z + src1.z;  
dest.w = src0.w + src1.w;
```

Instruction Information

Requirement	Value
Minimum operating system	Windows 98

Related topics

[Pixel Shader Instructions](#)

bem - ps

Article • 06/30/2021 • 2 minutes to read

Apply a fake bump environment-map transform.

Syntax

```
bem dst.rg, src0, src1
```

where

- dst.rg dst is the destination register. The red and green component write mask must be used.
- src0 is a source register.
- src1 is a source register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
bem					x				

This instruction performs the following calculation.

```
(Given n == dest register #)
dest.r = src0.r + D3DTSS_BUMPENVMAT00(stage n) * src1.r
          + D3DTSS_BUMPENVMAT10(stage n) * src1.g

dest.g = src0.g + D3DTSS_BUMPENVMAT01(stage n) * src1.r
          + D3DTSS_BUMPENVMAT11(stage n) * src1.g
```

Rules for using bem:

1. bem must appear in the first phase of a shader (that is, before a phase marker).
2. bem consumes two arithmetic instruction slots.
3. Only one use of this instruction is allowed per shader.

4. Destination writemask must be .rg /xy.
5. This instruction cannot be co-issued.
6. Aside from the restriction that destination write mask be .rg, modifiers on source src0, src1, and instruction modifiers are unconstrained.

Instruction Information

Requirement	Value
Minimum operating system	Windows 98

Related topics

[Pixel Shader Instructions](#)

break - ps

Article • 06/30/2021 • 2 minutes to read

Break out of the current loop at the nearest [endloop - ps](#) or [endrep - ps](#).

Syntax

```
break
```

Remarks

This instruction is supported in the following versions.

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
break					x	x	x	x	x

Requirement	Value
Minimum operating system	Windows 98

Related topics

[Pixel Shader Instructions](#)

break_comp - ps

Article • 11/20/2019 • 2 minutes to read

Break out of the current loop at the nearest [endloop - ps](#) or [endrep - ps](#), based on a per-component comparison.

Syntax

```
break_comp src0, src1
```

Where:

- _comp is a scalar (or single) comparison between the two source registers. It can be one of the following:

Syntax	Comparison
_gt	Greater than
_lt	Less than
_ge	Greater than or equal
_le	Less than or equal
_eq	Equal to
_ne	Not equal to

- src0 is a source register. Replicate swizzle is required if selecting a single component.
- src1 is a source register. Replicate swizzle is required if selecting a single component.

Remarks

This instruction is supported in the following versions.

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
break_comp					x	x	x	x	x

When the comparison is true, it breaks out of the current loop, as shown.

```
if (!(src0 comparison src1))
    jump to the corresponding endloop or endrep instruction;
```

Related topics

[Pixel Shader Instructions](#)

breakp - ps

Article • 11/20/2019 • 2 minutes to read

Conditionally break out of the current loop at the nearest [endloop - ps](#) or [endrep - ps](#). Use one of the components of the predicate register as a condition to determine whether or not to perform the instruction.

Syntax

```
breakp [!]p0.{x|y|z|w}
```

Where:

- [!] is an optional negate modifier.
- p0 is the [Predicate Register](#).
- {x|y|z|w} is the required replicate swizzle on p0.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
breakp					x	x	x	x	x

Related topics

[Pixel Shader Instructions](#)

call - ps

Article • 03/09/2021 • 2 minutes to read

Performs a function call to the instruction marked with the provided label.

Syntax

```
call l#
```

Where:

- l# is a [label - ps](#) marking the beginning of the subroutine to be called.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
call					x	x	x	x	x

This instruction does the following:

1. Push address of the next instruction to the return address stack.
2. Continue execution from the instruction marked by the label.

Related topics

[Pixel Shader Instructions](#)

callnz bool - ps

Article • 03/09/2021 • 2 minutes to read

Call if not zero. Performs a conditional call to the instruction marked by the label index.

Syntax

```
callnz l#, [!]b#
```

Where:

- l# is a [label - ps](#) marking the beginning of the subroutine to be called.
- [!] is an optional negate modifier.
- b# identifies a [Constant Boolean Register](#).

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
callnz bool					x	x	x	x	x

This instruction does the following:

```
if (specified Boolean register is not zero)
{
    Push address of the next instruction to the return address stack
    Continue execution from the instruction marked by the label
}
```

Related topics

[Pixel Shader Instructions](#)

callnz pred - ps

Article • 11/20/2019 • 2 minutes to read

Call with a predicate, if not zero. Performs a conditional call to the instruction marked by the label index. Predication uses a Boolean value to determine whether or not to perform the instruction.

Syntax

```
callnz l#, [!]p0.{x|y|z|w}
```

Where:

- where l# is a [label - ps](#) marking the beginning of the subroutine to be called.
- [!] is an optional negate modifier.
- p0 is the predicate register. See [Predicate Register](#).
- {x|y|z|w} is the required replicate swizzle on p0.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
callnz pred					x	x	x	x	x

This instruction does the following:

```
if (specified register component is not zero)
{
    Push address of the next instruction to the return address stack
    Continue execution from the instruction marked by the label
}
```

Related topics

[Pixel Shader Instructions](#)

cmp - ps

Article • 11/20/2019 • 2 minutes to read

Choose src1 if $\text{src0} \geq 0$. Otherwise, choose src2. The comparison is done per channel.

Syntax

```
cmp dst, src0, src1, src2
```

where

- dst is the destination register.
- src0 is a source register.
- src1 is a source register.
- src2 is a source register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
cmp		x	x	x	x	x	x	x	x

There are a few additional limitations for versions 1_2 and 1_3:

- Each shader can use up to a maximum of three cmp instructions.
- The destination register cannot be the same as any of the source registers.

This example does a four-channel comparison.

```
ps_1_4
def c0, -0.6, 0.6, 0, 0.6
def c1 0,0,0,0
def c2 1,1,1,1

mov r1, c1
mov r2, c2

cmp r0, c0, r1, r2 // r0 is assigned 1,0,0,0 based on the following:
```

```
// r0.x = c2.x because c0.x < 0  
// r0.y = c1.y because c0.y >= 0  
// r0.z = c1.z because c0.z >= 0  
// r0.w = c1.w because c0.w >= 0
```

Related topics

[Pixel Shader Instructions](#)

cnd - ps

Article • 11/20/2019 • 2 minutes to read

Conditionally chooses between src1 and src2, based on the comparison $\text{src0} > 0.5$.

Syntax

```
cnd dst, src0, src1, src2
```

where

- dst is the destination register.
- src0 is a source register.
- src1 is a source register.
- src2 is a source register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
cnd	x	x	x	x					

For versions 1_1 to 1_3, src0 must be r0.a.

```
// Version 1_1 to 1_3
if (r0.a > 0.5)
    dest = src1
else
    dest = src2
```

Version 1_4 compares each channel separately.

```
for each component in src0
{
    if (src0.component > 0.5)
```

```

    dest.component = src1.component
else
    dest.component = src2.component
}

```

These examples show a four-channel comparison done in a version 1_4 shader, as well as a single-channel comparison possible in a version 1_1 shader.

```

ps_1_4
def c0, -0.5, 0.5, 0, 0.6
def c1, 0,0,0,0
def c2, 1,1,1,1

cnd r1, c0, c1, c2 // r0 contains 1,1,1,0 because
// r1.x = c2.x because c0.x <= 0.5
// r1.y = c2.y because c0.y <= 0.5
// r1.z = c2.z because c0.z <= 0.5
// r1.w = c1.w because c0.w > 0.5

```

Version 1_1 to 1_3 compares against the replicated alpha channel of r0 only.

```

ps_1_1
def c0, -0.5, 0.5, 0, 0.6
def c1, 0,0,0,0
def c2, 1,1,1,1
mov r0, c0
cnd r1, r0.a, c1, c2 // r1 gets assigned 0,0,0,0 because
// r0.a > 0.5, therefore r1.xyzw = c1.xyzw

```

This example compares two values, A and B. The example assumes A is loaded into v0 and B is loaded into v1. Both A and B must be in the range of -1 to +1, and since the color registers (v_n) are defined to be between 0 and 1, the restriction happens to be satisfied in this example.

```

ps_1_1          // Version instruction
sub r0, v0, v1_bias // r0 = A - (B - 0.5)
cnd r0, r0.a, c0, c1 // r0 = ( A > B ? c0 : c1 )

// r0 = c0 if A > B, otherwise, r0 = c1

```

Related topics

[Pixel Shader Instructions](#)

crs - ps

Article • 03/09/2021 • 2 minutes to read

Computes a cross product using the right-hand rule.

Syntax

```
crs dst, src0, src1
```

where

- dst is the destination register.
- src0 is a source register.
- src1 is a source register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
crs					x	x	x	x	x

This instruction works as shown here.

```
dest.x = src0.y * src1.z - src0.z * src1.y;  
dest.y = src0.z * src1.x - src0.x * src1.z;  
dest.z = src0.x * src1.y - src0.y * src1.x;
```

Some restrictions on use:

- src0 cannot be the same register as dest.
- src1 cannot be the same register as dest.
- src0 cannot have any swizzle other than the default swizzle (.xyzw).
- src1 cannot have any swizzle other than the default swizzle (.xyzw).
- dest has to have exactly one of the following seven masks: .x | .y | .z | .xy | .xz | .yz | .xyz.
- dest must be a temporary register.

- dest must not be the same register as src0 or src1

Related topics

[Pixel Shader Instructions](#)

dcl - (sm2, sm3 - ps asm)

Article • 06/30/2021 • 2 minutes to read

Declare a pixel shader input register.

Syntax

```
dcl[_pp] dest[.mask]
```

Where:

- [_pp] is optional partial precision. See [Partial Precision](#).
- dest is a destination register. It must be either a [Input Color Register](#) (vn), or an [Texture Coordinate Register](#) (tn).
- [.mask] is an optional write mask that controls which components of the destination register that can be written to. See [Destination Register Write Mask](#).

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
dcl					x	x	x	x	x

All dcl instructions must appear before the first executable instruction.

Related topics

[Pixel Shader Instructions](#)

dcl_samplerType (sm2, sm3 - ps asm)

Article • 06/30/2021 • 2 minutes to read

Declare a pixel shader sampler.

Syntax

dcl_samplerType s#

where:

- _samplerType defines the sampler data type. This determines how many coordinates are required by each texture coordinate when sampling. The following texture coordinate dimensions are defined.
 - _2d
 - _cube
 - _volume
- s# identifies a sampler where s is an abbreviation for the sampler, and # is the sampler number. Samplers are pseudo registers because you cannot directly read or write to them.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
dcl_samplerType					x	x	x	x	x

All dcl_samplerType instructions must appear before the first executable instruction.

Example

```
dcl_cube t0.rgb; // Define a 3D texture map.  
  
add r0, r0, t0; // Perturb texture coordinates.  
texld r0, s0, r0; // Load r0 with a color sampled from stage0
```

```
// at perturbed texture coordinates r0.  
// This is a dependent texture read.
```

Related topics

[Pixel Shader Instructions](#)

dcl_semantics (sm3 - ps asm)

Article • 06/30/2021 • 2 minutes to read

Declare the association between vertex shader output and pixel shader input.

Syntax

```
dcl_semantics [_centroid] dst[.write_mask]
```

Where:

- `_semantics`: Identifies the intended data usage, and may be any of the values in [D3DDECLUSAGE](#) (without the D3DDECLUSAGE_ prefix). Additionally, an integer index can be appended to the semantic to distinguish parameters that use similar semantics.
- `[_Centroid]` is an optional instruction modifier. It is supported on `dcl_usage` instructions that declare the input registers and on texture lookup instructions. The centroid is appended with no space.
- `dst`: destination register. See [ps_3_0 Registers](#).
- `write_mask`: The same output register may be declared multiple times, each time with a unique write mask (so different semantics can be applied to individual components). However, the same semantic cannot be used multiple times in a declaration. This means that vectors must be four components or less, and cannot go across four-component register boundaries (individual output registers). When the `_psize` semantic is used, it should have a full write mask because it is considered a scalar. When the `_position` semantic is used, it should have full write mask because all four components have to be written.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
dcl_usage						x	x		

All `dcl_usage` instructions must appear before the first executable instruction.

Declaration Examples

```
ps_3_0

; Declaring inputs
dcl_normal      v0.xyz
dcl_blendweight v0.w ; Must be same reg# as normal, matching vshader packing
dcl_texcoord1   v1.y ; Mask can be any subset of mask from vshader semantic
dcl_texcoord0   v1.zw; Has to be same reg# as texcoord1, to match vshader

; Declaring samplers
dcl_2d s0
dcl_2d s1

def c0, 0, 0, 0, 0

mov r0.x, v1.y ; texcoord1
mov r0.y, c0
texld r0, r0, s0

texld r1, v1.zw, s1
...
(output regs in ps_3_0 are same as ps_2_0: oC0-oC3, oDepth)
```

Related topics

[Pixel Shader Instructions](#)

[Antialias Sample ↗](#)

def - ps

Article • 08/19/2020 • 2 minutes to read

Defines pixel shader floating-point constants.

Syntax

```
def dst, fValue1, fValue2, fValue3, fValue4
```

Where:

- dst is the destination register.
- fValue1 to fValue4 are floating-point values..

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
def	x	x	x	x	x	x	x	x	x

There are two ways to set a floating-point constant in a pixel shader.

1. Use def to define the constant directly inside a shader.
2. Use the API to set a constant with [SetPixelShaderConstantF](#).

def defines a shader constant whose value is loaded any time a shader is set to a device. These are called immediate constants. Immediate constants take precedence over constants set by the API method.

- Must appear before the first arithmetic or addressing instruction in shader.
- Can be intermixed with [dcl - \(sm2, sm3 - ps asm\)](#) instructions (which are the other type of instruction that resides at the beginning of a shader).
- dst register must be a [constant register](#).
- Write mask must be full (default).
- If a [constant register](#) is defined multiple times in a shader, the last one persists.

Related topics

Pixel Shader Instructions

defb - ps

Article • 08/19/2020 • 2 minutes to read

Defines a boolean constant value, which should be loaded any time a shader is set to a device.

Syntax

```
defb dest, booleanValue
```

where

- dst is the destination register.
- booleanValue is a single boolean value, either true or false.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
defb						x	x	x	x

The defb instruction defines a boolean shader constant whose value is loaded anytime a shader is set to a device. These are called immediate constants. Immediate constants take precedence over constants set by the API method SetPixelShaderConstantB.

There are two ways to set a booleanconstant in a shader.

1. Use defb to define the constant directly inside a shader.
2. Use the API methods to set a constant.
 - Use [SetPixelShaderConstantB](#) to set a Boolean constant.

Related topics

[Pixel Shader Instructions](#)

[def - ps](#)

defi - ps

defi - ps

Article • 08/19/2020 • 2 minutes to read

Defines an integer constant value, which should be loaded any time a shader is set to a device.

Syntax

```
defi dst, integerValue
```

- dst is the destination register.
- integerValue is a constant integer value.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
defi					x	x	x	x	x

The defi instruction defines a shader constant whose value is loaded anytime a shader is set to a device. These are called immediate constants. Immediate constants take precedence over constants set by the API method SetPixelShaderConstantB.

There are two ways to set a constant in a shader.

1. Use defi to define the constant directly inside a shader.
2. Use the API methods to set a constant.
 - Use [SetPixelShaderConstantB](#) to set a Boolean constant.
 - Use [SetPixelShaderConstantF](#) to set a floating-point constant.
 - Use [SetPixelShaderConstantI](#) to set an integer constant.

Related topics

[Pixel Shader Instructions](#)

dp2add - ps

Article • 11/20/2019 • 2 minutes to read

Performs a 2D dot product and a scalar addition.

Syntax

```
dp2add dst, src0, src1, src2.{x|y|z|w}
```

Where:

- dst is a destination register.
- src0, src1, and src2 are three source registers.
- {x|y|z|w} is the required replicate swizzle on src2.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
dp2add					x	x	x	x	x

The scalar value for add is chosen by the replicate swizzle on src2.

The following code snippet shows the operations performed.

```
dest = src0.r * src1.r + src0.g * src1.g + src2.replicate_swizzle  
// The scalar result is replicated to the write mask components
```

Related topics

[Pixel Shader Instructions](#)

dp3 - ps

Article • 03/09/2021 • 2 minutes to read

Computes the three-component dot product of the source registers.

Syntax

```
dp3 dst, src0, src1
```

where

- dst is the destination register.
- src0 is a source register.
- src1 is a source register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
dp3	x	x	x	x	x	x	x	x	x

The following code snippet shows the operations performed:

```
dest.x = dest.y = dest.z = dest.w =
(src0.x * src1.x) + (src0.y * src1.y) + (src0.z * src1.z);
```

This instruction runs in the vector pipeline, always writing out to the color channels. For version 1_4, this instruction still uses the vector pipeline but may write to any channel.

An instruction with a destination register .rgb (.xyz) write mask may be co-issued with dp3 As shown below.

```
dp3 r0.rgb, t0, v0          // Copy scalar result to color components
+mov r2.a, t0                // Copy alpha component from t0 in parallel
```

The dp3 instruction can be modified using the [Source Register Signed Scaling](#) input argument modifier (_bx2) applied to its input arguments if they are not already expanded to signed dynamic range. For a lighting shader, the saturate instruction modifier (_sat) is often used to clamp the negative values to black, as shown in the following example.

```
dp3_sat r0, t0_bx2, v0_bx2    // t0 is a bump map, v0 contains the light  
direction
```

Related topics

[Pixel Shader Instructions](#)

dp4 - ps

Article • 03/09/2021 • 2 minutes to read

Computes the four-component dot product of the source registers.

Syntax

```
dp4 dst, src0, src1
```

where

- dst is the destination register.
- src0 is a source register.
- src1 is a source register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
dp4		x	x	x	x	x	x	x	x

The following code snippet shows the operations performed:

```
dest.x = dest.y = dest.z = dest.w =
    (src0.x * src1.x) + (src0.y * src1.y) +
    (src0.z * src1.z) + (src0.w * src1.w);
```

Limitations for ps_1_2 and ps_1_3:

- Each shader can use up to a maximum of four dp4 instructions.
- Each dp4 instruction counts as two arithmetic instructions.

Limitations for 1_X versions:

- This instruction cannot be co-issued because dp4 runs in both the vector and alpha pipeline.

Related topics

[Pixel Shader Instructions](#)

dsx - ps

Article • 11/20/2019 • 2 minutes to read

Compute the rate of change in the render target's x-direction.

Syntax

```
dsx dst, src
```

Where:

- dst is a destination register.
- src is an input source register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
dsx					x	x	x	x	x

The rate of change computed from the source register is an approximation on the contents of the same register in adjacent pixel(s) running the pixel shader in lock-step with the current pixel.

The dsx And [dsy](#) instructions compute their result by looking at the current contents of the source register (per component) for the various pixels in the local area executing in the lock-step. The exact formula used to compute the gradient varies depending on the hardware but should be consistent with the way the hardware does the same operations as part of the level-of-detail calculation process for texture sampling.

Related topics

[Pixel Shader Instructions](#)

[texldd - ps](#)

dsy - ps

Article • 11/20/2019 • 2 minutes to read

Compute the rate of change in the render target's y-direction.

Syntax

```
dsy dst, src
```

Where:

- dst is a destination register.
- src is an input source register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
dsy					x	x	x	x	x

The rate of change computed from the source register is an approximation on the contents of the same register in adjacent pixel(s) running the pixel shader in lock-step with the current pixel.

The [dsx](#) And dsy instructions compute their result by looking at the current contents of the source register (per component) for the various pixels in the local area executing in the lock-step. The exact formula used to compute the gradient varies depending on the hardware but should be consistent with the way the hardware does the same operations as part of the level-of-detail calculation process for texture sampling.

Related topics

[Pixel Shader Instructions](#)

[texldd - ps](#)

else - ps

Article • 03/09/2021 • 2 minutes to read

Start of an else block.

Syntax

```
else
```

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
else					x	x	x	x	x

If the condition in the corresponding [if](#) statement is true, the code enclosed by the if statement and the matching else is run.

Related topics

[Pixel Shader Instructions](#)

[if bool - ps](#)

[endif - ps](#)

endif - ps

Article • 11/20/2019 • 2 minutes to read

Marks the end of an [if...else](#) block.

Syntax

```
endif
```

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
endif					x	x	x	x	x

Related topics

[Pixel Shader Instructions](#)

[if bool - ps](#)

[else - ps](#)

endloop - ps

Article • 11/20/2019 • 2 minutes to read

End of a [loop - ps](#)...endloop block.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
endloop							x	x	

endloop must follow the last instruction of a [loop - ps](#) block.

Example

```
loop aL, i3
    add r1, r0, c2[ aL ]
endloop
```

Related topics

[Pixel Shader Instructions](#)

endrep - ps

Article • 11/20/2019 • 2 minutes to read

End a [rep - ps](#)...endrep block.

Syntax

```
endrep
```

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
endrep					x	x	x	x	x

Example

```
rep i2  
    add r0, r0, c0  
endrep
```

Related topics

[Pixel Shader Instructions](#)

exp - ps

Article • 03/09/2021 • 2 minutes to read

Provides full precision exponential 2^x .

Syntax

```
exp dst, src
```

where

- dst is the destination register.
- src is a source register. Source register requires explicit use of replicate swizzle; that is, exactly one of the .x, .y, .z, .w swizzle components (or the .r, .g, .b, .a equivalents) must be specified. See [Source Register Swizzling](#).

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
exp					x	x	x	x	x

The following code snippet shows the operations performed:

```
dest.x = dest.y = dest.z = dest.w = (float)pow(2,  
src.replicateSwizzleComponent);
```

Related topics

[Pixel Shader Instructions](#)

frc - ps

Article • 03/09/2021 • 2 minutes to read

Returns the fractional portion of each input component.

Syntax

```
frc dst, src
```

where

- dst is the destination register.
- src is a source register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
frc					x	x	x	x	x

The following code snippet shows conceptually how the instruction operates.

```
dest.x = src.x - (float)floor(src.x);
dest.y = src.y - (float)floor(src.y);
dest.z = src.z - (float)floor(src.z);
dest.w = src.w - (float)floor(src.w);
```

The floor function converts the argument passed in to the greatest integer that is less than (or equal to) the argument. This is converted to a float and then subtracted from the original value. The resulting fractional value ranges from 0.0 through 1.0.

Related topics

[Pixel Shader Instructions](#)

if bool - ps

Article • 11/20/2019 • 2 minutes to read

Start of an if block.

Syntax

```
if bool
```

Where:

- bool is a bool (Boolean) register number. See [Constant Boolean Register](#).

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
if bool					x	x	x	x	x

If the source Boolean register in the if statement is true, the code enclosed by the if statement and the matching [endif - ps](#) or [else - ps](#) is executed. Otherwise, the code enclosed by the else - ps...endif - ps statements is executed. This instruction consumes one instruction slot.

An if block can be nested.

An if block cannot straddle a loop block.

An if block can be followed by a statement block, and/or an [else - ps](#) instruction, and/or an [endif - ps](#) instruction.

Example

This instruction provides conditional static flow control.

```
defb b3, true

if b3
// Instructions to run if b3 is nonzero
else
// Instructions to run otherwise
endif
```

Related topics

[Pixel Shader Instructions](#)

[else - ps](#)

[endif - ps](#)

if_comp - ps

Article • 11/20/2019 • 2 minutes to read

Start an `if bool - ps...else - ps...endif - ps` block, with a condition based on values that could be computed in a shader. This instruction is used to skip a block of code, based on a condition.

Syntax

```
if_comp src0, src1
```

Where:

- `_comp` is a comparison between the two source registers. It can be one of the following:

Syntax	Comparison
<code>_gt</code>	Greater than
<code>_lt</code>	Less than
<code>_ge</code>	Greater than or equal
<code>_le</code>	Less than or equal
<code>_eq</code>	Equal to
<code>_ne</code>	Not equal to

- `src0` is a source register. Replicate swizzle is required to select a component.
- `src1` is a source register. Replicate swizzle is required to select a component.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
<code>if_comp</code>					x	x	x	x	x

This instruction is used to skip a block of code, based on a condition.

```
if (src0 comparison src1)
    jump to the corresponding else or endif instruction;
```

Be careful using the equals and not equals comparison modes on floating point numbers. Because rounding occurs during floating point calculations, the comparison can be done against an epsilon value (small nonzero number) to avoid errors.

Restrictions include:

- if_comp...[else - ps](#)...[endif - ps](#) blocks (along with the predicated [if](#) blocks) can be nested up to 24 layers deep.
- src0 and src1 registers require a replicate swizzle.
- if_comp blocks must end with an [else - vs](#) or [endif - vs](#) instruction.
- if_comp...[else - ps](#)...[endif - ps](#) blocks cannot straddle a loop block. The if_comp block must be completely inside or outside the loop block.

Example

This instruction provides conditional dynamic flow control.

```
if_lt r3.x, r4.y
// Instructions to run if r3.x < r4.y

else
// Instructions to run otherwise

endif
```

Related topics

[Pixel Shader Instructions](#)

if pred - ps

Article • 11/20/2019 • 2 minutes to read

Start of an [if bool - ps...else - ps...endif - ps](#) block, with the condition taken from the content of the predicate register.

Syntax

```
if [!]pred.replicateSwizzle
```

Where:

- [!] is an optional NOT modifier. This modifies the value in the predicate register.
- pred is the [Predicate Register](#).
- replicateSwizzle is a single component that is copied (or replicated) to all four components (swizzled). Valid components are: [x, y, z, w] or [r, g, b, a].

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
if_pred					x	x	x	x	x

This instruction is used to skip a block of code, based on a channel of the predicate register. Each if_pred block must end with an [else - ps](#) or [endif - ps](#) instruction.

Restrictions include:

if_pred blocks can be nested. This counts to the total dynamic nesting depth along with [if_comp](#) blocks.

An if_pred block cannot straddle a loop block; it should either be completely inside it or surround it.

Related topics

[Pixel Shader Instructions](#)

label - ps

Article • 03/09/2021 • 2 minutes to read

Mark the next instruction as having a label index.

Syntax

label l#

where # identifies the label number.

For ps_2_x, the label number must be between between 0 and 15.

For ps_2_sw, ps_3_0, and ps_3_sw, the label number must be between between 0 and 2047.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
label					x	x	x	x	x

This instruction defines a label located at the next shader instruction. The label instruction can only be present directly after a [ret](#) instruction (which defines the end of the previous subroutine or main program).

Related topics

[Pixel Shader Instructions](#)

loop - ps

Article • 11/20/2019 • 2 minutes to read

Starts a loop...[endloop - ps](#) block.

Syntax

```
loop aL, i#
```

Where:

- aL is the [Loop Counter Register](#) holding the current loop count.
- i# is a [Constant Integer Register](#). See remarks.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
loop						x	x		

- The [Loop Counter Register](#) (aL) holds the current loop count and can be used for relative addressing into [Input Color Register](#) (v#) inside the loop block.
- i#.x specifies the iteration count. The legal range is [0, 255]. Note that this instruction does not increment or decrement the value of i#.x.
- i#.y specifies the initial value of the [Loop Counter Register](#) (aL) register. The legal range is [0, 255]. Note that this instruction does not increment or decrement the value of i#.y.
- i#.z specifies the step/stride size. The legal range is [-128, 127].
- i#.w is not used by the loop block and has to be 0.
- Loop blocks may be nested. See [Flow Control Limitations](#).
- When nested, the value of the [Loop Counter Register](#) (aL) refers to the immediate enclosing loop block.
- Loop blocks are allowed to be either completely inside an if* block or completely surrounding it. No straddling is allowed.

Example

```
loop aL, i3
    add r1, r0, v2[ aL ]
endloop
```

Related topics

[Pixel Shader Instructions](#)

log - ps

Article • 03/09/2021 • 2 minutes to read

Full precision $\log_2(x)$.

Syntax

log dst, src

where

- dst is the destination register.
- src is a source register. Source register requires explicit use of replicate swizzle; that is, exactly one of the .x, .y, .z, .w swizzle components (or the .r, .g, .b, .a equivalents) must be specified.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
log					x	x	x	x	x

The following code snippet shows the operations performed.

```
float v = abs(src);
if (v != 0)
{
    dest.x = dest.y = dest.z = dest.w =
        (float)(log(v)/log(2));
}
else
{
    dest.x = dest.y = dest.z = dest.w = -FLT_MAX;
}
```

This instruction accepts a scalar source whose sign bit is ignored. The result is replicated to all four channels.

Related topics

[Pixel Shader Instructions](#)

lrp - ps

Article • 03/09/2021 • 2 minutes to read

Interpolates linearly between the second and third source registers by a proportion specified in the first source register.

Syntax

```
lrp dst, src0, src1, src2
```

where

- dst is the destination register.
- src0 is a source register.
- src1 is a source register.
- src2 is a source register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
lrp	x	x	x	x	x	x	x	x	x

This instruction performs the linear interpolation based on the following formula.

```
dest = src0 * src1 + (1-src0) * src2  
// which is the same as  
dest = src2 + src0 * (src1 - src2)
```

Related topics

[Pixel Shader Instructions](#)

m3x2 - ps

Article • 03/09/2021 • 2 minutes to read

Multiplies a 3-component vector by a 3x2 matrix.

Syntax

```
m3x2 dst, src0, src1
```

where

- dst is the destination register. Result is a 2-component vector.
- src0 is a source register representing a 3-component vector.
- src1 is a source register representing a 3x2 matrix, which corresponds to the first of 2 consecutive registers.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
m3x2					x	x	x	x	x

The xy mask is required for the destination register. Negate and swizzle modifiers are allowed for src0 but not for src1.

The following equations show how the instruction operates:

```
dest.x = (src0.x*src1.x) + (src0.y*src1.y) + (src0.z*src1.z);  
dest.y = (src0.x*src2.x) + (src0.y*src2.y) + (src0.z*src2.z);
```

The input vector is in register src0. The input 3x2 matrix is in register src1 and the next higher register in the same register file, as shown in the following expansion. A 2D result is produced, leaving the other elements of the destination register (dest.z and dest.w) unaffected.

This operation is commonly used for 2D transforms. This instruction is implemented as a pair of dot products as shown here.

```
m3x2    r0.xy, r1, c0    which will be expanded to:
```

```
dp3    r0.x, r1, c0  
dp3    r0.y, r1, c1
```

Swizzle and negate modifiers are invalid for the src1 register. The dst and src0 register, or any of src1+i registers, cannot be the same.

Related topics

[Pixel Shader Instructions](#)

m3x3 - ps

Article • 03/09/2021 • 2 minutes to read

Multiplies a 3-component vector by a 3x3 matrix.

Syntax

```
m3x3 dst, src0, src1
```

where

- dst is the destination register. Result is a 3-component vector.
- src0 is a source register representing a 3-component vector.
- src1 is a source register representing a 3x3 matrix, which corresponds to the first of 3 consecutive registers.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
m3x3					x	x	x	x	x

The xyz mask is required for the destination register. Negate and swizzle modifiers are allowed for src0 but not for src1.

The following code snippet shows the operations performed.

```
dest.x = (src0.x * src1.x) + (src0.y * src1.y) + (src0.z * src1.z);
dest.y = (src0.x * src2.x) + (src0.y * src2.y) + (src0.z * src2.z);
dest.z = (src0.x * src3.x) + (src0.y * src3.y) + (src0.z * src3.z);
```

The input vector is in register src0. The input 3x3 matrix is in register src1, and the next two higher registers, as shown in the expansion below. A 3D result is produced, leaving the other element of the destination register (dest.w) unaffected.

This operation is commonly used for transforming normal vectors during lighting computations. This instruction is implemented as a set of dot products as shown below.

```
m3x3 r0.xyz, r1, c0 which will be expanded to:
```

```
dp3    r0.x, r1, c0
dp3    r0.y, r1, c1
dp3    r0.z, r1, c2
```

Related topics

[Pixel Shader Instructions](#)

m3x4 - ps

Article • 03/09/2021 • 2 minutes to read

Multiplies a 3-component vector by a 3x4 matrix.

Syntax

```
m3x4 dst, src0, src1
```

where

- dst is the destination register. Result is a 4-component vector.
- src0 is a source register representing a 3-component vector.
- src1 is a source register representing a 3x4 matrix, which corresponds to the first of 4 consecutive registers.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
m3x4					x	x	x	x	x

The xyzw (default) mask is required for the destination register. Negate and swizzle modifiers are allowed for src0 but not for src1.

The following code snippet shows the operations performed.

```
dest.x = (src0.x*src1.x) + (src0.y*src1.y) + (src0.z*src1.z);
dest.y = (src0.x*src2.x) + (src0.y*src2.y) + (src0.z*src2.z);
dest.z = (src0.x*src3.x) + (src0.y*src3.y) + (src0.z*src3.z);
dest.w = (src0.x*src4.x) + (src0.y*src4.y) + (src0.z*src4.z);
```

The input vector is in register src0. The input 3x4 matrix is in register src1, and the next two higher registers, as shown in the expansion below.

This operation is commonly used for transforming a position vector by a matrix that has a projective effect but applies no translation. This instruction is implemented as a pair of dot products as shown here.

```
m3x4    r0.xyzw, r1, c0    will be expanded to:
```

```
dp3 r0.x, r1, c0  
dp3 r0.y, r1, c1  
dp3 r0.z, r1, c2  
dp3 r0.w, r1, c3
```

Related topics

[Pixel Shader Instructions](#)

m4x3 - ps

Article • 03/09/2021 • 2 minutes to read

Multiplies a 4-component vector by a 4x3 matrix.

Syntax

```
m4x3 dst, src0, src1
```

where

- dst is the destination register. Result is a 3-component vector.
- src0 is a source register representing a 4-component vector.
- src1 is a source register representing a 4x3 matrix, which corresponds to the first of 3 consecutive registers.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
m4x3					x	x	x	x	x

The xyz mask is required for the destination register. Negate and swizzle modifiers are allowed for src0, but not for src1.

The following code snippet shows the operations performed.

```
dest.x = (src0.x*src1.x) + (src0.y*src1.y) + (src0.z*src1.z) +
(src0.w*src1.w);
dest.y = (src0.x*src2.x) + (src0.y*src2.y) + (src0.z*src2.z) +
(src0.w*src2.w);
dest.z = (src0.x*src3.x) + (src0.y*src3.y) + (src0.z*src3.z) +
(src0.w*src3.w);
```

The input vector is in register src0. The input 4x3 matrix is in register src1, and the next two higher registers, as shown in the expansion below. A 3D result is produced, leaving

the other element of the destination register (dest.w) unaffected.

This operation is commonly used for transforming a position vector by a matrix that has no projective effect, such as occurs in model-space transformations. This instruction is implemented as a set of dot products as shown below.

```
m4x3    r0.xyz, r1, c0    will be expanded to:
```

```
dp4    r0.x, r1, c0  
dp4    r0.y, r1, c1  
dp4    r0.z, r1, c2
```

Swizzle and negate modifiers are invalid for the src1 register. The dst and src0 register cannot be the same.

Related topics

[Pixel Shader Instructions](#)

m4x4 - ps

Article • 03/09/2021 • 2 minutes to read

Multiplies a 4-component vector by a 4x4 matrix.

Syntax

```
m4x4 dst, src0, src1
```

where

- dst is the destination register. Result is a 4-component vector.
- src0 is a source register representing a 4-component vector.
- src1 is a source register representing a 4x4 matrix, which corresponds to the first of 4 consecutive registers.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
m4x4					x	x	x	x	x

The xyzw (default) mask is required for the destination register. Negate and swizzle modifiers are allowed for src0, but not for src1.

Swizzle and negate modifiers are invalid for the src0 register. The dst and src0 registers cannot be the same.

The following code snippet shows the operations performed.

```
dest.x = (src0.x*src1.x) + (src0.y*src1.y) + (src0.z*src1.z) +
(src0.w*src1.w);
dest.y = (src0.x*src2.x) + (src0.y*src2.y) + (src0.z*src2.z) +
(src0.w*src2.w);
dest.z = (src0.x*src3.x) + (src0.y*src3.y) + (src0.z*src3.z) +
(src0.w*src3.w);
```

```
dest.w = (src0.x*src4.x) + (src0.y*src4.y) + (src0.z*src4.z) +  
(src0.w*src4.w);
```

The input vector is in register src0. The input 4x4 matrix is in register src1, and the next three higher registers, as shown in the expansion below.

This operation is commonly used for transforming a position by a projection matrix. This instruction is implemented as a set of dot products as shown here.

m4x4 r0.xyzw, r1, c0 will be expanded to:

```
dp4 r0.x, r1, c0  
dp4 r0.y, r1, c1  
dp4 r0.z, r1, c2  
dp4 r0.w, r1, c3
```

Related topics

[Pixel Shader Instructions](#)

mad - ps

Article • 11/20/2019 • 2 minutes to read

Multiply and add instruction. Sets the destination register to $(src0 * src1) + src2$.

Syntax

```
mad dst, src0, src1, src2
```

where

- dst is the destination register.
- src0 is a source register.
- src1 is a source register.
- src2 is a source register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
mad	x	x	x	x	x	x	x	x	x

The following code snippet shows the operations performed.

```
dest.x = src0.x * src1.x + src2.x;  
dest.y = src0.y * src1.y + src2.y;  
dest.z = src0.z * src1.z + src2.z;  
dest.w = src0.w * src1.w + src2.w;
```

Related topics

[Pixel Shader Instructions](#)

mov - ps

Article • 11/20/2019 • 2 minutes to read

Move data between registers.

Syntax

```
mov dst, src
```

where

- dst is the destination register.
- src is a source register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
mov	x	x	x	x	x	x	x	x	x

Related topics

[Pixel Shader Instructions](#)

mul - ps

Article • 03/09/2021 • 2 minutes to read

Multiplies sources into the destination.

Syntax

```
mul dst, src0, src1
```

where

- dst is the destination register.
- src0 is a source register.
- src1 is a source register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
mul	x	x	x	x	x	x	x	x	x

The following code snippet shows the operations performed.

```
dest.x = src0.x * src1.x;
dest.y = src0.y * src1.y;
dest.z = src0.z * src1.z;
dest.w = src0.w * src1.w;
```

Related topics

[Pixel Shader Instructions](#)

max - ps

Article • 03/09/2021 • 2 minutes to read

Calculates the maximum of the sources.

Syntax

```
max dst, src0, src1
```

where

- dst is the destination register.
- src0 is a source register.
- src1 is a source register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
max					x	x	x	x	x

The following code snippet shows the operations performed.

```
dest.x=(src0.x >= src1.x) ? src0.x : src1.x;
dest.y=(src0.y >= src1.y) ? src0.y : src1.y;
dest.z=(src0.z >= src1.z) ? src0.z : src1.z;
dest.w=(src0.w >= src1.w) ? src0.w : src1.w;
```

Related topics

[Pixel Shader Instructions](#)

min - ps

Article • 03/09/2021 • 2 minutes to read

Calculates the minimum of the sources.

Syntax

```
min dst, src0, src1
```

where

- dst is the destination register.
- src0 is a source register.
- src1 is a source register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
min					x	x	x	x	x

The following code snippet shows the operations performed.

```
dest.x=(src0.x < src1.x) ? src0.x : src1.x;
dest.y=(src0.y < src1.y) ? src0.y : src1.y;
dest.z=(src0.z < src1.z) ? src0.z : src1.z;
dest.w=(src0.w < src1.w) ? src0.w : src1.w;
```

Related topics

[Pixel Shader Instructions](#)

nrm - ps

Article • 03/09/2021 • 2 minutes to read

Normalize a 3D vector.

Syntax

```
nrm dst, src
```

where

- dst is the destination register.
- src is a source register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
nrm					x	x	x	x	x

This instruction works conceptually as shown here.

$$\text{squareRootOfTheSum} = (\text{src0.x} * \text{src0.x} + \text{src0.y} * \text{src0.y} + \text{src0.z} * \text{src0.z})^{1/2};$$

```
dest.x = src0.x * (1 / squareRootOfTheSum);
dest.y = src0.y * (1 / squareRootOfTheSum);
dest.z = src0.z * (1 / squareRootOfTheSum);
dest.w = src0.w * (1 / squareRootOfTheSum);
```

The dest and src registers cannot be the same. The dest register must be a temporary register.

This instruction performs the linear interpolation based on the following formula.

```
float f = src0.x*src0.x + src0.y*src0.y + src0.z*src0.z;
if (f != 0)
    f = (float)(1/sqrt(f));
else
    f = FLT_MAX;

dest.x = src0.x*f;
dest.y = src0.y*f;
dest.z = src0.z*f;
dest.w = src0.w*f;
```

Related topics

[Pixel Shader Instructions](#)

nop - ps

Article • 03/09/2021 • 2 minutes to read

No operation is performed.

Syntax

```
nop
```

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
nop	x	x	x	x	x	x	x	x	x

This instruction performs a no-op, or no operation. The syntax for calling it is as follows:

```
nop
```

Related topics

[Pixel Shader Instructions](#)

phase - ps

Article • 11/20/2019 • 2 minutes to read

The phase instruction marks the transition between phase 1 and phase 2. If no phase instruction is present, the entire shader runs as if it is a phase 2 shader.

This instruction applies to version 1_4 only.

Syntax

```
phase
```

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
phase					x				

Shader instructions that occur before the phase instruction are phase 1 instructions. All other instructions are phase 2 instructions. By having two phases for instructions, the maximum number of instructions per shader is increased.

The unfortunate side-effect of the phase transition is that the alpha component of [temporary registers](#) do not persist across the transition. In other words, the alpha component must be reinitialized after the phase instruction.

Example

This example shows how to group instructions as phase 1 or phase 2 instructions within a shader.

The phase instruction is also commonly called the phase marker because it marks the transition between phase 1 and 2 instructions. In a version 1_4 pixel shader, if the phase marker is not present, the shader is run as if it is running in phase 2. This is important because there are differences between phase 1 and 2 instructions and register availability. The differences are noted throughout the reference section.

```
ps_1_4
// Add phase 1 instructions here

phase
// Add phase 2 instructions here
```

Related topics

[Pixel Shader Instructions](#)

pow - ps

Article • 03/09/2021 • 2 minutes to read

Full precision $\text{abs}(\text{src0})^{\text{src1}}$.

Syntax

```
pow dst, src0, src1
```

where

- dst is the destination register.
- src0 is an input source register. Source register requires explicit use of replicate swizzle, that is, exactly one of the .x, .y, .z, .w swizzle components (or the .r, .g, .b, .a equivalents) must be specified.
- src1 is an input source register. Source register requires explicit use of replicate swizzle, that is, exactly one of the .x, .y, .z, .w swizzle components (or the .r, .g, .b, .a equivalents) must be specified.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
pow					x	x	x	x	x

This instruction works as follows:

$$\text{dest.x} = \text{dest.y} = \text{dest.z} = \text{dest.w} = [\text{abs}(\text{src0})]^{\text{src1}};$$

This is a scalar instruction, therefore the source registers should have replicate swizzles to indicate which channels are used.

The input power (src1) must be a scalar.

The scalar result is replicated to all four output channels.

This instruction could be expanded as $\exp(\text{src1} * \log(\text{src0}))$.

The dst register should be a temporary register, and should not be the same register as src1.

Related topics

[Pixel Shader Instructions](#)

ps

Article • 11/20/2019 • 2 minutes to read

This instruction specifies the shader version number and works on all shader versions.

Syntax

```
ps_mainVer_subVer
```

Input Arguments

Input arguments contain a single main version number with a single sub version number. The allowable combinations are listed in the table below.

Main versions	Sub versions
1	1, 2, 3, 4
2	0, x (extended), sw (software)
3	0, sw (software)

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
ps	x	x	x	x	x	x	x	x	x

This instruction must be the first non-comment instruction in a pixel shader.

Hardware accelerated versions of the software (versions without _sw in the version number), can process vertices with hardware acceleration or use software vertex processing. Software versions (versions with _sw in the version number) process vertices only with software.

Examples

This partial example declares a version 1_1 pixel shader.

```
ps_1_1
```

This partial example declares a version 1_4 pixel shader.

```
ps_1_4
```

Related topics

[Pixel Shader Instructions](#)

rcp - ps

Article • 03/09/2021 • 2 minutes to read

Computes the reciprocal of the source scalar.

Syntax

```
rcp dst, src
```

where

- dst is the destination register.
- src is a source register. Source register requires explicit use of replicate swizzle, that is, exactly one of the .x, .y, .z, .w swizzle components (or the .r, .g, .b, .a equivalents) must be specified.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
rcp					x	x	x	x	x

The output must be exactly 1.0 if the input is exactly 1.0. A source of 0.0 yields infinity.

The scalar result is replicated to all channels in the destination write mask.

Precision should be at least $1.0/(2^{22})$ absolute error over the range (1.0, 2.0) because common implementations will separate mantissa and exponent.

Related topics

[Pixel Shader Instructions](#)

rep - ps

Article • 11/20/2019 • 2 minutes to read

Start a rep...endrep - ps block.

Syntax

```
rep i#
```

where i# is an integer register that specifies the repeat count in the .x component. See [Constant Integer Register](#).

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
rep					x	x	x	x	x

- i#.x specifies the iteration count. The legal range is [0, 255]. Note that this instruction does not increment or decrement the value of i#.x.
- i#.yzw are not used by the repeat block.
- Repeat blocks may be nested. See [Flow Control Limitations](#).
- Repeat blocks are allowed to be either completely inside an if* block or completely surrounding it. No straddling is allowed.
- Using the same i# for different or nested rep instructions is fine - each loop will iterate based on the specified count.

Example

```
rep i2  
    add r0, r0, c0  
endrep
```

Related topics

[Pixel Shader Instructions](#)

ret - ps

Article • 11/20/2019 • 2 minutes to read

Takes the address of an instruction from the return address stack and continues execution from it. In the case of the main function, this instruction stops shader execution.

Syntax

```
ret
```

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
ret					x	x	x	x	x

This instruction takes the address of an instruction from the return address stack and continues execution from it. In the case of the main function, this instruction stops shader execution.

The ret instruction consumes one vertex shader instruction slot.

If a shader contains no subroutines, using ret at the end of the main program is optional.

Multiple return statements are not permitted in the main program or in any subroutine, the first return statement is treated as the end of the main program or subroutine.

Related topics

[Pixel Shader Instructions](#)

rsq - ps

Article • 03/09/2021 • 2 minutes to read

Computes the reciprocal square root (positive only) of the source scalar.

Syntax

```
rsq dst, src
```

where

- dst is the destination register.
- src is a source register. Source register requires explicit use of replicate swizzle, that is, exactly one of the .x, .y, .z, .w swizzle components (or the .r, .g, .b, .a equivalents) must be specified.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
rsq					x	x	x	x	x

The absolute value is taken before processing.

Precision should be at least $1.0/(2^{22})$ absolute error over the range (1.0, 4.0) because common implementations will separate mantissa and exponent.

The output must be exactly 1.0 if the input is exactly 1.0. A source of 0.0 yields infinity.

Related topics

[Pixel Shader Instructions](#)

setp_comp - ps

Article • 03/09/2021 • 2 minutes to read

Set the predicate register.

Syntax

```
setp_comp dst, src0, src1
```

Where:

- _comp is a per-channel comparison between the two source registers. It can be one of the following:

Syntax	Comparison
_gt	Greater than
_lt	Less than
_ge	Greater than or equal
_le	Less than or equal
_eq	Equal to
_ne	Not equal to

- dst is the [Predicate Register](#) register, p0.
- src0 is a source register.
- src1 is a source register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
setp_comp					x	x	x	x	x

This instruction operates as:

```
per channel in destination write mask
{
    dst.channel = src0.channel cmp src1.channel
}
```

For each channel that can be written according to the destination write mask, save the boolean result of the comparison operation between the corresponding channels of src0 and src1 (after the source modifier swizzles have been resolved).

Source registers allow arbitrary component swizzles to be specified.

The destination register allows arbitrary write masks.

The dst register must be the predicate register.

Applying the Predicate Register

Once the predicate register has been initialized with setp_comp, it can be used to control an instruction per component. Here's the syntax:

```
([!]p0[.swizzle]) instruction dest, src0, ...
```

Where:

- [!] is an optional boolean NOT
- p0 is the predicate register
- [.swizzle] is an optional swizzle to apply to the contents of the predicate register before using it to mask the instruction. The available swizzles are: .xyzw (default when none specified), or any replicate swizzle: .x/.r, .y/.g, .z/.b or .a/.w.
- instruction is any arithmetic, or texture instruction. This cannot be a static or dynamic flow control instruction.
- dest, src0, ... are the registers required by the instruction

Assuming the predicate register has been set up with (true, true, false, false) component values, it can be applied to this instruction:

```
(p0) add r1, r2, r3
```

to perform a 2 component add.

```
r1.x = r2.x + r3.x  
r1.y = r2.y + r3.y
```

The z and w components of r1 will not be written since the predicate register contained false in components z and w.

Applying the predicate register to an arithmetic or texture instruction increases its instruction slot count by 1.

The predicate register can also be applied to [if pred - ps](#), [callnz pred - ps](#) and [breakp - ps](#) instructions. These flow control instructions do not have any increase in the instruction slot count when using the predicate register.

Related topics

[Pixel Shader Instructions](#)

sincos - ps

Article • 03/09/2021 • 2 minutes to read

Computes sine and cosine, in radians.

Syntax

ps_2_0 and ps_2_x

```
sincos dst.{x|y|xy}, src0.{x|y|z|w}, src1, src2
```

Where:

- dst is the destination register and has to be a [Temporary Register](#) (r#). The destination register must have exactly one of the following three masks: .x | .y | .xy.
- src0 is a source register that provides the input angle, which must be within [-pi, +pi]. {x|y|z|w} is the required replicate swizzle.
- src1 and src2 are source registers and must be two different [Constant Float Registers](#) (c#). The values of src1 and src2 must be that of the macros [D3DSINCOSCONST1](#) and [D3DSINCOSCONST2](#) respectively.

ps_3_0

```
sincos dst.{x|y|xy}, src0.{x|y|z|w}
```

Where:

- dst is a destination register and has to be a [Temporary Register](#) (r#). The destination register must have exactly one of the following three masks: .x | .y | .xy.
- src0 is a source register that provides the input angle, which must be within [-pi, +pi]. {x|y|z|w} is the required replicate swizzle.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
-----------------------	-----	-----	-----	-----	-----	-----	------	-----	------

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
sincos					x	x	x	x	x

ps_2_0 and ps_2_x

For ps_2_0 and ps_2_x, sincos can be used with predication, but with one restriction to the swizzle of the [Predicate Register](#) (p0): only replicate swizzle (.x | .y | .z | .w) is allowed.

For ps_2_0 and ps_2_x, the instruction operates as follows (V = the scalar value from src0 with a replicate swizzle):

- If the write mask is .x:

```
dest.x = cos(V)
dest.y is undefined when the instruction completes
dest.z is undefined when the instruction completes
dest.w is not touched by the instruction
```

- If the write mask is .y:

```
dest.x is undefined when the instruction completes
dest.y = sin(V)
dest.z is undefined when the instruction completes
dest.w is not touched by the instruction
```

- If the write mask is .xy:

```
dest.x = cos(V)
dest.y = sin(V)
dest.z is undefined when the instruction completes
dest.w is not touched by the instruction
```

ps_3_0

For ps_3_0, sincos can be used with predication without any restriction. See [Predicate Register](#).

For ps_3_0, the instruction operates as follows (V = the scalar value from src0 with a replicate swizzle):

- If the write mask is .x:

```
dest.x = cos(V)
dest.y is not touched by the instruction
dest.z is not touched by the instruction
dest.w is not touched by the instruction
```

- If the write mask is .y:

```
dest.x is not touched by the instruction
dest.y = sin(V)
dest.z is not touched by the instruction
dest.w is not touched by the instruction
```

- If the write mask is .xy:

```
dest.x = cos(V)
dest.y = sin(V)
dest.z is not touched by the instruction
dest.w is not touched by the instruction
```

The application can map an arbitrary angle (in radians) to the range $[-\pi, +\pi]$ using the following shader pseudocode:

```
def c0, pi, 0.5, 2*pi, 1/(2*pi)
mad r0.x, input_angle, c0.w, c0.y
frc r0.x, r0.x
mad r0.x, r0.x, c0.z, -c0.x
```

Related topics

[Pixel Shader Instructions](#)

sub - ps

Article • 03/09/2021 • 2 minutes to read

Subtracts sources.

Syntax

```
sub dst, src0, src1
```

where

- dst is the destination register.
- src0 is a source register.
- src1 is a source register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
sub	x	x	x	x	x	x	x	x	x

This instruction performs the subtraction shown in this formula.

```
dest = src0 - src1
```

Related topics

[Pixel Shader Instructions](#)

tex - ps

Article • 08/19/2020 • 2 minutes to read

Loads the destination register with color data (RGBA) sampled from a texture. The texture must be bound to a particular texture stage (n) using [SetTexture](#). Texture sampling is controlled by [SetSamplerState](#).

Syntax

```
tex dst
```

where

- dst is the destination register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
tex	x	x	x						

The destination register number specifies the texture stage number.

Texture sampling uses texture coordinates to look up, or sample, a color value at the specified (u,v,w,q) coordinates while taking into account the texture stage state attributes.

The texture coordinate data is interpolated from the vertex texture coordinate data and is associated with a specific texture stage. The default association is a one-to-one mapping between texture stage number and texture coordinate declaration order. This means that the first set of texture coordinates defined in the vertex format are by default associated with texture stage 0.

Texture coordinates may be associated with any stage using two techniques. When using a fixed function vertex shader or the fixed function pipeline, the texture stage state flag TSS_TEXCOORDINDEX can be used in [SetTextureStageState](#) to associate

coordinates to a stage. Otherwise, the texture coordinates are output by the vertex shader oT_n registers when using a programmable vertex shader.

Related topics

[Pixel Shader Instructions](#)

texbem - ps

Article • 08/19/2020 • 2 minutes to read

Apply a fake bump environment-map transform. This is accomplished by modifying the texture address data of the destination register, using address perturbation data (du,dv), and a 2D bump environment matrix.

Syntax

```
texbem dst, src
```

where

- dst is the destination register.
- src is a source register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
texbem	x	x	x						

The red and green color data in the src register is interpreted as the perturbation data (du,dv).

This instruction transforms red and green components in the source register using the 2D bump environment-mapping matrix. The result is added to the texture coordinate set corresponding to the destination register number, and is used to sample the current texture stage.

This operation always interprets du and dv as signed quantities. For versions 1_0 and 1_1, the [Source Register Signed Scaling](#) input modifier (_bx2) is not permitted on the input argument.

This instruction produces defined results when input textures contain signed format data. Mixed format data works only if the first two channels contain signed data. For more information about surface formats, see [D3DFORMAT](#).

This can be used for a variety of techniques based on address perturbation, including fake per-pixel environment mapping and diffuse lighting (bump mapping).

When using this instruction, texture registers must follow the following sequence.

```
// The texture assigned to stage t(n) contains the (du,dv) data  
// The texture assigned to stage t(m) is sampled  
tex      t(n)  
texbem  t(m),  t(n)      where m > n
```

The calculations done within the instruction are shown below.

```
// 1. New values for texture addresses (u',v') are calculated  
// 2. Sample the texture using (u',v')
```

$u' = \text{TextureCoordinates(stage m)}_u + \text{D3DTSS_BUMPENVMAT00}(\text{stage m}) * t(n)_R + \text{D3DTSS_BUMPENVMAT10}(\text{stage m}) * t(n)_G$

$v' = \text{TextureCoordinates(stage m)}_v + \text{D3DTSS_BUMPENVMAT01}(\text{stage m}) * t(n)_R + \text{D3DTSS_BUMPENVMAT11}(\text{stage m}) * t(n)_G$

$t(m)_{\text{RGBA}} = \text{TextureSample}(\text{stage m})$

// using (u',v') as coordinates

Register data that has been read by a texbem - ps or [texbeml - ps](#) instruction cannot be read later, except by another texbem - ps or texbeml - ps.

```
// This example demonstrates the validation error caused by t0 being reread:  
ps_1_1  
tex t0  
texbem t1, t0  
add r0, t1, t0  
  
(Instruction Error) (Statement 4) Register data that has been read by  
texbem or texbeml instruction cannot be read by other instructions
```

Examples

Here is an example shader with the texture maps identified and the texture stages identified.

```
ps_1_1
tex t0           ; Define t0 to get a 2-tuple DuDv
texbem t1, t0    ; Compute (u',v')
                  ; Sample t1 using (u',v')
mov r0, t1       ; Output result
```

texbem requires the following textures in the following texture stages.

- Stage 0 is assigned a bump map with (du, dv) perturbation data.
- Stage 1 uses a texture map with color data.
- This instruction sets the matrix data on the texture stage that is sampled.
- This is different from the functionality of the fixed function pipeline where the perturbation data and the matrices occupy the same texture stage.

Related topics

[Pixel Shader Instructions](#)

texbeml - ps

Article • 08/19/2020 • 2 minutes to read

Apply a fake bump environment-map transform with luminance correction. This is accomplished by modifying the texture address data of the destination register, using address perturbation data (du,dv), a 2D bump environment matrix, and luminance.

Syntax

```
texbeml dst, src
```

where

- dst is the destination register.
- src is a source register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
texbeml	x	x	x						

The red and green color data in the src register is interpreted as the perturbation data (du,dv). The blue color data in the src register is interpreted as the luminance data.

This instruction transforms the red and green components in the source register using the 2D bump environment mapping matrix. The result is added to the texture coordinate set corresponding to the destination register number. A luminance correction is applied using the luminance value and the bias texture stage values. The result is used to sample the current texture stage.

This can be used for a variety of techniques based on address perturbation such as fake per-pixel environment mapping.

This operation always interprets du and dv as signed quantities. For versions 1_0 and 1_1, the [Source Register Signed Scaling](#) input modifier (_bx2) is not permitted on the input argument.

This instruction produces defined results when input textures contain mixed format data. For more information about surface formats, see [D3DFORMAT](#).

```
// When using this instruction, texture registers must follow  
//   the following sequence:  
// The texture assigned to stage tn contains the (du,dv) data  
// The texture assigned to stage t(m) is sampled  
tex      t(n)  
texbeml t(m),  t(n)      where m > n
```

This example shows the calculations done within the instruction.

```
// 1. New values for texture addresses (u',v') are calculated  
// 2. Sample the texture using (u',v')  
// 3. Luminance correction is applied
```

$u' = \text{TextureCoordinates(stage m)}_u +$
 $D3DTSS_BUMPENVMAT00(\text{stage m}) * t(n)_R +$
 $D3DTSS_BUMPENVMAT10(\text{stage m}) * t(n)_G$

$v' = \text{TextureCoordinates(stage m)}_v +$
 $D3DTSS_BUMPENVMAT01(\text{stage m}) * t(n)_R +$
 $D3DTSS_BUMPENVMAT11(\text{stage m}) * t(n)_G$

$t(m)_{\text{RGBA}} = \text{TextureSample}(\text{stage m}) \text{ using } (u', v')$ as coordinates

$t(m)_{\text{RGBA}} = t(m)_{\text{RGBA}} * [(t(n)_B * D3DTSS_BUMPENVLSCALE(\text{stage m})) + D3DTSS_BUMPENVLOFFSET(\text{stage m})]$

Register data that has been read by a [texbem](#) or [texbeml](#) instruction cannot be read later, except by another [texbem](#) or [texbeml](#).

```
// This example demonstrates the validation error caused by  
//   t0 being reread  
ps_1_1  
tex t0
```

```
texbem t1, t0  
add r0, t1, t0
```

(Instruction Error) (Statement 4) Register data that has been read by texbem or texbeml instruction cannot be read by other instructions

Examples

Here is an example shader with the texture maps identified and the texture stages identified.

```
ps_1_1  
tex t0           ; Define t0 to get a 2-tuple DuDv  
texbeml t1, t0   ; Compute (u',v')  
                  ; Apply luminance correction  
                  ; Sample t1 using (u',v')  
mov r0, t1       ; Output result
```

This example requires the following textures in the following texture stages.

- Stage 0 is assigned a bump map with (du, dv) perturbation data.
- Stage 1 is assigned a texture map with color data.
- texbeml sets the matrix data on the texture stage that is sampled. This is different from the functionality of the fixed function pipeline where the perturbation data and the matrices occupy the same texture stage.

Related topics

[Pixel Shader Instructions](#)

texcoord - ps

Article • 08/19/2020 • 2 minutes to read

Interprets texture coordinate data (UVW1) as color data (RGBA).

Syntax

```
texcoord dst
```

where

- dst is the destination register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
texcoord	x	x	x						

This instruction interprets the texture coordinate set (UVW1) corresponding to the destination register number as color data (RGBA). If the texture coordinate set contains fewer than three components, the missing components are set to 0. The fourth component is always set to 1. All values are clamped between 0 and 1.

The advantage of texcoord is that it provides a way to pass vertex data interpolated at high precision directly into the pixel shader. However, when the data is written into the destination register, some precision will be lost, depending on the number of bits used by the hardware for registers.

No texture is sampled by this instruction. Only texture coordinates set on this texture stage are relevant.

Any texture data (such as position, normal, and light source direction) can be mapped by a vertex shader into a texture coordinate. This is done by associating a texture with a texture register using [SetTexture](#) and by specifying how the texture sampling is done using [SetTextureStageState](#). If the fixed function pipeline is used, be sure to supply the TSS_TEXCOORDINDEX flag.

This instruction is used as follows:

```
texcoord tn
```

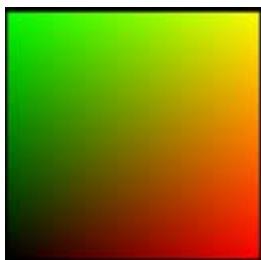
An [Texture Coordinate Register](#) (tn) contains four color values (RGBA). The data can also be thought of as vector data (xyzw). texcoord will retrieve three of these values (xyz) from texture coordinate set x, and the fourth component (w) is set to 1. The texture address is copied from the texture coordinate set n. The result is clamped between 0 and 1.

This example is for illustration only. The C code accompanying the shader has not been optimized for performance.

Here is an example shader using texcoord.

```
ps_1_1      ; version instruction
texcoord t0  ; declare t0 hold texture coordinates,
              ; which represent rgba values in this example
mov r0, t0  ; move the color in t0 to output register r0
```

The rendered output from the pixel shader is shown in the following illustration. The (u,v,w,1) coordinate values map to the (rgb) channels. The alpha channel is set to 1. At the corners of the illustration, coordinate (0,0,0,1) is interpreted as black; (1,0,0,1) is red; (0,1,0,1) is green; and (1,1,0,1) contains green and red, producing yellow.



Additional code is required to use this shader and an example scenario is shown below.

```
// This code creates the shader from a file. The contents of
// the shader file can also be supplied as a text string.
LPD3DXBUFFER      pCode;

// Assemble the vertex shader from the file
D3DXAssembleShaderFromFile(strPShaderPath, 0, NULL, &pCode, NULL);
m_pd3dDevice->CreatePixelShader((DWORD*)pCode->GetBufferPointer(),
```

```
&m_hPixelShader);  
pCode->Release();  
  
// This code defines the object vertex data  
struct CUSTOMVERTEX  
{  
    FLOAT x, y, z;  
    FLOAT tu1, tv1;  
};  
  
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3DFVF_TEX1|TEXCOORD2(0))  
  
static CUSTOMVERTEX g_Vertices[]=  
{  
    //  x      y      z      u1      v1  
    { -1.0f, -1.0f, 0.0f, 0.0f, 0.0f, },  
    { +1.0f, -1.0f, 0.0f, 1.0f, 0.0f, },  
    { +1.0f, +1.0f, 0.0f, 1.0f, 1.0f, },  
    { -1.0f, +1.0f, 0.0f, 0.0f, 1.0f, },  
};
```

Related topics

[Pixel Shader Instructions](#)

texcrd - ps

Article • 08/19/2020 • 2 minutes to read

Copies texture coordinate data from the source texture coordinate iterator register as color data in the destination register.

Syntax

```
texcrd dst, src
```

where

- dst is the destination register.
- src is a source register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
texcrd					x				

This instruction interprets coordinate data as color data (RGBA).

No texture is sampled by this instruction. Only texture coordinates set on this texture stage are relevant.

When using texcrd, keep in mind the following detail about how data is copied from the source register to the destination register. The source texture coordinate register (t#) holds data in the range [-D3DCAPS9.MaxTextureRepeat, D3DCAPS9.MaxTextureRepeat], while the destination register (r#) can hold data only in the (likely smaller) range [-D3DCAPS9.PixelShader1xMaxValue, D3DCAPS9.PixelShader1xMaxValue]. Note that for pixel shader version 1_4, D3DCAPS9.PixelShader1xMaxValue must be a minimum of eight. The texcrd instruction, in the process of clamping source data that is out of range of the destination register, is likely to behave differently on different hardware. The first pixel shader version 1_4 hardware on the market will perform a special clamp for values outside of range. This clamp is designed to produce a number that can fit into the destination register, but also to preserve texture addressing behavior for out-of-range

data (see [D3DTEXTUREADDRESS](#)) if the data were to be subsequently used for texture sampling. However, new hardware from different manufacturers might not exhibit this behavior and might just chop data to fit the destination register range. Therefore, the safest course of action when using pixel shader version 1_4 texcrd is to supply texture coordinate data only into the pixel shader that is already within the range [-8,8] so that you do not rely on the way hardware clamps.

Unlike texcoord_, texcrd does not clamp values between 0 and 1.

Rules for using texcrd :

1. The same .xyz or .xyw modifier must be applied to every read of an individual t(n) register within a texcrd or texld instruction.
2. The fourth channel result of texcrd is unset/undefined in all cases.
3. The third channel is unset/undefined for the xyw_dw case.

Examples

The complete set of allowed syntax for texcrd, taking into account all valid source modifier/selector and destination write mask combinations, is shown below. Note that the .rgba and .xyzw notation can be used interchangeably.

Copies first three channels of texture coordinate iterator register, t(n), into r(m). The fourth channel of r(m) is uninitialized.

```
texcrd r(m).rgb, t(n).xyz  
// Produces the same result as the previous instruction  
texcrd r(m).rgb, t(n)
```

Puts first, second, and fourth components of t(n) into first three channels of r(m). The fourth channel of r(m) is uninitialized.

```
texcrd r(m).rgb, t(n).xyw
```

Here is a projective divide example using the _dw modifier.

This example copies x/w and y/w from t(n) into the first two channels of r(m). The third and fourth channels of r(m) are uninitialized. Any data previously written to the third

channel of $r(m)$ will be lost. Data in the fourth channel of $r(m)$ is lost due to the phase marker. For version 1_4, the D3DTFF_PROJECTED flag is ignored.

```
texcrd r(m).rg, t(n)_dw.xyw
```

Related topics

[Pixel Shader Instructions](#)

texdepth - ps

Article • 11/20/2019 • 2 minutes to read

Calculate depth values to be used in the depth buffer comparison test for this pixel.

Syntax

```
texdepth dst
```

where

- dst is the destination register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
texdepth					x				

This instruction uses r5.r / r5.g in the depth buffer comparison test for this pixel. The data in the blue and alpha channels is ignored. If r5.g = 0, the result of r5.r / r5.g = 1.0.

Temporary register r5 is the only register that this instruction can use.

After executing this instruction, temporary register r5 is unavailable for additional use in the shader.

When multisampling, using this instruction eliminates most of the benefit of the higher resolution depth buffer. Because the pixel shader runs once per pixel, the single depth value output by [texm3x2depth - ps](#) or texdepth will be used for each of the subpixel depth comparison tests.

Examples

Here is an example using texdepth.

```
ps_1_4
texld r0, t0           // Sample texture from texture stage 0 (dest
                      // register number) into r0
                      // Use texture coordinate data from t0
texcrd r1.rgb, t1     // Load a second set of texture coordinate data into r1
add    r5.rg, r0, r1  // Add the two sets of texture coordinate data
phase   phase          // Phase marker, required when using texdepth
instruction
texdepth r5           // Calculate pixel depth as r5.r / r5.g
                      // Do other color calculations with shader output r0
```

Related topics

[Pixel Shader Instructions](#)

texdp3 - ps

Article • 11/20/2019 • 2 minutes to read

Performs a three-component dot product between data in the texture register number and the texture coordinate set corresponding to the destination register number.

Syntax

```
texdp3 dst, src
```

where

- dst is the destination register.
- src is a source register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
texdp3		x	x						

Texture registers must use the following sequence.

```
tex    t(n)      // Define tn as a standard 3-vector (tn must be
                  // defined in some way before texdp3 uses it)
texdp3 t(m), t(n) // where m > n
                  // Perform a three-component dot product between tn and
                  // the texture coordinate set m. The scalar result is
                  // replicated to all components of t(m)
```

Here is more detail about how the dot product is accomplished.

The texdp3 instruction performs a three-component dot product and replicates it to all four color channels.

$$t(m)_{\text{RGBA}} = \text{TextureCoordinates(stage } m \text{)}_{\text{UVW}} * t(n)_{\text{RGB}}$$

Related topics

[Pixel Shader Instructions](#)

texdp3tex - ps

Article • 11/20/2019 • 2 minutes to read

Performs a three-component dot product and uses the result to do a 1D texture lookup.

Syntax

```
texdp3tex dst, src
```

where

- dst is the destination register.
- src is a source register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
texdp3tex		x	x						

Texture registers must use the following sequence.

```
tex      t(n)      // Define tn as a standard 3-vector (tn must be
                   // defined in some way before texdp3tex uses it)
texdp3tex t(m), t(n) // where m > n
                     // Perform a three-component dot product between t(n)
and
                     // the texture coordinate set m. Use the scalar result
to
                     // do a 1D texture lookup at texturestage m and place
                     // the result in t(m)
```

Here is more detail about how the dot product and texture lookup are done.

The texdp3tex instruction performs a three-component dot product.

$$u' = \text{TextureCoordinates(stage m)}_{UVW} * t(n)_{RGB}$$

The result is used to sample the texture at texture stage m by performing a 1D lookup.

$t(m)_{\text{RGBA}} = \text{TextureSample}(\text{stage } m)_{\text{RGBA}}$ using $(u', 0, 0)$ as coordinates

Related topics

[Pixel Shader Instructions](#)

texkill - ps

Article • 11/20/2019 • 2 minutes to read

Cancels rendering of the current pixel if any of the first three components (UVW) of the texture coordinates is less than zero.

Syntax

```
texkill dst
```

where

- dst is a destination register

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
texkill	x	x	x	x	x	x	x	x	x

This instruction corresponds to the HLSL's [clip](#) function.

texkill does not sample any texture. It operates on the first three components of the texture coordinates given by the destination register number. For ps_1_4, texkill operates on the data in the first three components of the destination register.

You can use this instruction to implement arbitrary clip planes in the rasterizer.

When using vertex shaders, the application is responsible for applying the perspective transform. This can cause problems for the arbitrary clipping planes because if it contains anisomorphic scale factors, the clip planes need to be transformed as well. Therefore, it is best to provide an unprojected vertex position to use in the arbitrary clipper, which is the texture coordinate set identified by the texkill operator.

This instruction is used as follows:

texkill tn // The pixel masking is accomplished as follows: if (the x,y,z components of TextureCoordinates(stage n).uvw < 0) cancel pixel render

For pixel shader 1_1, 1_2, and 1_3, texkill operates on the texture coordinate set given by the destination register number. In version 1_4, however, texkill operates on the data contained in the [Texture Coordinate Register](#) (tn) or in the temporary register (rn) that has been specified as the destination.

When multisampling is enabled, any antialiasing effect achieved on polygon edges due to multisampling will not be achieved along any edge that has been generated by texkill. The pixel shader runs once per pixel.

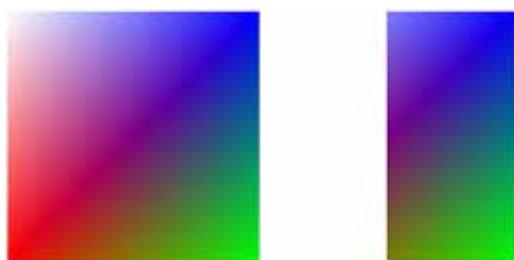
This example is for illustration only.

This example masks out pixels that have negative texture coordinates. The pixel colors are interpolated from vertex colors provided in the vertex data.

```
ps_1_1      // Version instruction
texkill t0    // Mask out pixel using texture coordinates from stage 0
mov r0, v0    // Move the diffuse color in v0 to r0

// The rendered output from the pixel shader is shown below
```

The texture coordinates range from -0.5 to 0.5 in u, and 0.0 to 1.0 in v. This instruction causes the negative u values to get masked out. The first illustration below shows the vertex color applied to the quad without the texkill instruction applied. The second illustration below shows the result of the texkill instruction. The pixel colors from the texture coordinates below 0 (where x goes from -0.5 to 0.0) are masked out. The background color (white) is used where the pixel color is masked.



The texture coordinate data is declared in the vertex data declaration in this example.

```
struct CUSTOMVERTEX
{
    FLOAT x, y, z;
    DWORD color;
    FLOAT tu1, tv1;
};
```

```
#define D3DFVF_CUSTOMVERTEX  
(D3DFVF_XYZ|D3DFVF_DIFFUSE|D3DFVF_TEX1|D3DTEXCOORD2(0))  
  
static CUSTOMVERTEX g_Vertices[]=  
{  
    //  x      y      z      color          u1,      v1  
    { -1.0f, -1.0f, 0.0f, 0xffff0000, -0.5f, 1.0f, },  
    { 1.0f, -1.0f, 0.0f, 0xff00ff00, 0.5f, 1.0f, },  
    { 1.0f,  1.0f, 0.0f, 0xff0000ff, 0.5f, 0.0f, },  
    { -1.0f,  1.0f, 0.0f, 0xffffffff, -0.5f, 0.0f, },  
};
```

Related topics

[Pixel Shader Instructions](#)

texId - ps_1_4

Article • 06/30/2021 • 2 minutes to read

Loads the destination register with color data (RGBA) sampled using the contents of the source register as texture coordinates. The sampled texture is the texture associated with the destination register number.

texId dst, src

Registers

Value	Description	v_n	c_n	t_n	r_n	Pixel shader version
dst	Destination register			x		1_4
src	Source register			x		1_4 phase 1
				x	x	1_4 phase

When using r(n) as a source register, the first three components (XYZ) must have been initialized in the previous phase of the shader.

To learn more about registers, see [ps_1_1_ps_1_2_ps_1_3_ps_1_4 Registers](#).

Remarks

This instruction samples the texture in the texture stage associated with the destination register number. The texture is sampled using texture coordinate data from the source register.

The syntax for the texId and texcrd instructions expose support for a projective divide with a Texture Register Modifier. For pixel shader version 1.4, the D3DTTFF_PROJECTED texture transform flag is always ignored.

Rules for using texId:

1. The same .xyz or .xyw modifier must be applied to every read of an individual t(n) register within both texcrd or texId instructions. If .xyw is being used on t(n)

register reads, this can be mixed with other reads of the same t(n) register using .xyw_dw.

2. The _dz source modifier is only valid on texld with r(n) source register (thus phase 2 only).
3. The _dz source modifier may be used no more than two times per shader.

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
texld					x				

Examples

The texld instruction offers some control over which components of the source texture coordinate data are used. The complete set of allowed syntax for texld follows, and includes all valid source register modifiers, selectors, and write mask combinations.

```
texld r(m), t(n).xyz  
// Uses xyz from t(n) to sample 1D, 2D, or 3D texture
```

```
texld r(m), t(n)  
// Same as previous
```

```
texld r(m), t(n).xyw  
// Uses xyw (skipping z) from t(n) to sample 1D, 2D or 3D texture
```

```
texld r(m), t(n)_dw.xyw  
// Samples 1D or 2D texture at x/w, y/w from t(n). The result  
// is undefined for a cube-map lookup.
```

```
texld r(m), r(n).xyz  
// Samples 1D, 2D, or 3D texture at xyz from r(m)  
// This is possible in the second phase of the shader
```

```
texld r(m), r(n)
// Same as previous
```

```
texld r(m), r(n)_dz.xyz
// Samples 1D or 2D texture at x/z, y/z from r(m)
// Possible only in second phase
// The result is undefined for a cube-map lookup
```

```
texld r(n), r(n)_dz
// Same as previous
```

Related topics

[Pixel Shader Instructions](#)

texId - ps_2_0 and up

Article • 03/31/2021 • 2 minutes to read

Sample a texture at a particular sampler, using provided texture coordinates. This instruction is different from the [texId - ps_1_4](#) instruction used in pixel shader version 1_4.

Syntax

```
texId dst, src0, src1
```

Where:

- dst is a destination register.
- src0 is a source register that provides the texture coordinates for the texture sample.
- src1 identifies the [Sampler \(Direct3D 9 asm-ps\)](#) (s#), where # specifies which texture sampler number to sample. The sampler has associated with it a texture and a sampler state defined by [D3DSAMPLERSTATETYPE](#).

ps_2_0 and ps_2_x

dst must be a [Temporary Register](#) (r#) and only .xyzw mask (default mask) is allowed.

src0 must be either a [Texture Coordinate Register](#) (t#) or a [Temporary Register](#) (r#), with no modifier or swizzle.

src1 must be a [Sampler \(Direct3D 9 asm-ps\)](#) (s#), with no modifier or swizzle.

If the D3DD3DPSHADERCAPS2_0_NODEPENDENTREADLIMIT cap bit is not set (in D3DPSHADERCAPS2_0), a given texture instruction ([texId](#), [texldp](#), [texldb - ps](#), [texldd](#)) may be dependent upon, at most, third order. A first-order dependent texture instruction is a texture instruction in which either:

- src0 is a [Temporary Register](#) (r#).
- dst was previously written, now being written again.

A second-order dependent texture instruction is defined as a texture instruction that reads or writes to a [Temporary Register](#) (r#) whose contents, before executing the texture instruction, depend (perhaps indirectly) on the outcome of a first-order

dependent texture instruction. An $(n)^{th}$ -order dependent texture instruction derives from an $(n - 1)^{th}$ -order texture instruction.

ps_3_0

src1 must be a [Sampler \(Direct3D 9 asm-ps\)](#) (s#), with no modifier. Swizzle is allowed on src0 or src1. The swizzle is applied to the texture coordinates before texture lookup.

Remarks

This instruction is supported in the following versions:

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
texld					x	x	x	x	x

The number of coordinates required for src0 to perform the texture sample depends on how src1 was declared, plus the .w component. Sampler types are declared with [dcl_samplerType \(sm2, sm3 - ps asm\)](#). If src1 is declared as a 2D sampler, then src0 must contain .xy coordinates; if src1 is declared as either a cube sampler or a volume sampler, then src0 must contain .xyz coordinates. Sampling a texture with fewer dimensions than are present in the texture coordinate is allowed since the extra texture coordinate component(s) are ignored.

If the source texture contains fewer than four components, defaults are placed in the missing components. Defaults depend on the texture format as shown in the following table:

Texture Format	Default Values
D3DFMT_R5G6B5, D3DFMT_R8G8B8, D3DFMT_L8, D3DFMT_L16, D3DFMT_R3G3B2, D3DFMT_CxV8U8, D3DFMT_L6V5U5	A = 1.0
D3DFMT_V8U8, D3DFMT_V16U16, D3DFMT_G16R16, D3DFMT_G16R16F, D3DFMT_G32R32F	B = A = 1.0
D3DFMT_A8	R = G = B = 0.0
D3DFMT_R16F, D3DFMT_R32F	G = B = A = 1.0

Texture Format	Default Values
All depth/stencil formats	R = B = 0.0, A = 1.0

Related topics

[Pixel Shader Instructions](#)

texldb - ps

Article • 08/19/2020 • 2 minutes to read

Biased texture load instruction. This instruction uses the fourth element (.a or .w) to bias the texture-sampling level-of-detail just before sampling.

Syntax

```
texldb dst, src0, src1
```

Where:

- dst is the destination register.
- src0 is a source register that provides the texture coordinates for the texture sample. See [Texture Coordinate Register](#).
- src1 identifies the [Sampler \(Direct3D 9 asm-ps\)](#) (s#), where # specifies which texture sampler number to sample. The sampler has associated with it a texture and a sampler state defined by [D3DSAMPLERSTATETYPE](#).

For the restrictions when using texldb, see the [texld - ps_2_0 and up](#) instruction.

ps_2_0 and ps_2_x

dst must be a [Temporary Register](#) (r#) and only .xyzw mask (default mask) is allowed.

src0 must be either a [Texture Coordinate Register](#) (t#) or a [Temporary Register](#) (r#), with no modifier or swizzle.

src1 must be a [Sampler \(Direct3D 9 asm-ps\)](#) (s#), with no modifier or swizzle.

If the D3DD3DPSHADERCAPS2_0_NODEPENDENTREADLIMIT cap bit is not set (in D3DPSHADERCAPS2_0), a given texture instruction (texld, texldp, texldb, texldd) may be dependent upon, at most, third order. A first-order dependent texture instruction is a texture instruction in which either:

- src0 is a [Temporary Register](#) (r#).
- dst was previously written, now being written again.

A second-order dependent texture instruction is defined as a texture instruction that reads or writes to a [Temporary Register](#) (r#) whose contents, before executing the

texture instruction, depend (perhaps indirectly) on the outcome of a first-order dependent texture instruction. An $(n)^{th}$ -order dependent texture instruction derives from an $(n - 1)^{th}$ -order texture instruction.

ps_3_0

src1 must be a [Sampler \(Direct3D 9 asm-ps\)](#) (s#), with no modifier. Swizzle is allowed on src1, and when applied, the texture lookup results are pre-swizzled before written to dst.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
texldb					x	x	x	x	x

texldb biases the mipmap level-of-detail, computed normally as part of the sample process by the (signed) value in src0.w. Positive bias values will result in smaller mipmaps being selected and vice versa. For ps_2_0 and ps_2_x, bias values can be within the range [-3.0, +3.0]. For ps_3_0, bias values can be within the range [-16.0, +15.0]. Bias values outside these ranges produce undefined results. The sampler state D3DSAMP_MIPMAPLODBIAS is still honored, and the texldb bias is added to this, but on a per-pixel basis. After the biased level-of-detail is computed, D3DSAMP_MAXMIPLEVEL is still honored and the texture sample occurs. After texldb, the contents of src0 are unaffected (unless dst is the same register).

The number of coordinates required for src0 to perform the texture sample depends on how src1 was declared, plus the .w component. Sampler types are declared with [dcl_samplerType \(sm2, sm3 - ps asm\)](#). If src1 is declared as a 2D sampler, then src0 must contain .xyw coordinates; if src1 is declared as either a cube sampler or a volume sampler, then src0 must contain .xyzw coordinates. Sampling a 2D texture with .xyzw coordinates is allowed (the .z coordinate is ignored).

If the source texture contains fewer than four components, defaults are placed in the missing components. Defaults depend on the texture format as shown in the following table:

Texture Format	Default Values

Texture Format	Default Values
D3DFMT_R5G6B5, D3DFMT_R8G8B8, D3DFMT_L8, D3DFMT_L16, D3DFMT_R3G3B2, D3DFMT_CxV8U8, D3DFMT_L6V5U5	A = 1.0
D3DFMT_V8U8, D3DFMT_V16U16, D3DFMT_G16R16, D3DFMT_G16R16F, D3DFMT_G32R32F	B = A = 1.0
D3DFMT_A8	R = G = B = 0.0
D3DFMT_R16F, D3DFMT_R32F	G = B = A = 1.0
All depth/stencil formats	R = B = 0.0, A = 1.0

Related topics

[Pixel Shader Instructions](#)

texIdd - ps

Article • 08/19/2020 • 2 minutes to read

Samples a texture with additional gradient inputs.

Syntax

```
texIdd, dst, src0, src1, src2, src3
```

Where:

- dst is a destination register.
- src0 is a source register that provides the texture coordinates for the texture sample. See [Texture Coordinate Register](#).
- src1 identifies the source sampler register (s#), where # specifies which texture sampler number to sample. The sampler has associated with it a texture and a control state defined by the [D3DSAMPLERSTATETYPE](#) enumeration (ex. D3DSAMP_MINFILTER).
- src2 is an input source register that specifies the x gradient.
- src3 is an input source register that specifies the y gradient.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
texIdd					x *	x	x	x	x

* This instruction is only supported by ps_2_a. It is not supported by ps_2_b. For more information about profiles, see [D3DXGetPixelShaderProfile](#).

This instruction samples a texture using the texture coordinates at src0, the sampler specified by src1, and the gradients DSX and DSY coming from src2 and src3. The x and y gradient values are used to select the appropriate mipmap level of the texture for sampling.

All sources support arbitrary swizzles.

All write masks are valid on the destination.

Related topics

[Pixel Shader Instructions](#)

texldl - ps

Article • 03/09/2021 • 3 minutes to read

Sample a texture with a particular sampler. The particular mipmap level-of-detail being sampled has to be specified as the fourth component of the texture coordinate.

Syntax

```
texldl dst, src0, src1
```

Where:

- dst is a destination register.
- src0 is a source register that provides the texture coordinates for the texture sample. See [Texture Coordinate Register](#).
- src1 identifies the source sampler register (s#), where # specifies which texture sampler number to sample. The sampler has associated with it a texture and a control state defined by the [D3DSAMPLERSTATETYPE](#) enumeration (for example, D3DSAMP_MINFILTER).

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
texldl							x	x	

texldl looks up the texture set at the sampler stage referenced by src1. The level-of-detail is selected from src0.w. This value can be negative in which case the level-of-detail selected is the zeroth one (biggest map) with the MAGFILTER. Since src0.w is a floating point value, the fractional value is used to interpolate (if MIPFILTER is LINEAR) between two mip levels. Sampler states MIPMAPLODBIAS and MAXMIPLEVEL are honored. For more information about sampler states, see [D3DSAMPLERSTATETYPE](#).

If a shader program samples from a sampler that does not have a texture set, then 0001 is obtained in the destination register.

The following is a rough algorithm that the reference device follows:

```

LOD = src0.w + LODBIAS;
if (LOD <= 0 )
{
    LOD = 0;
    Filter = MagFilter;
    tex = Lookup( MAX(MAXMIPLEVEL, LOD), Filter );
}
else
{
    Filter = MinFilter;
    LOD = MAX( MAXMIPLEVEL, LOD );
    tex = Lookup( Floor(LOD), Filter );
    if( MipFilter == LINEAR )
    {
        tex1 = Lookup( Ceil(LOD), Filter );
        tex = (1 - frac(src0.w))*tex + frac(src0.w)*tex1;
    }
}

```

Restrictions:

- The texture coordinates should not be scaled by texture size.
- dst must be a [Temporary Register](#) (r#).
- dst can accept a writemask. See [Destination Register Write Mask](#).
- Defaults for missing components are either 0 or 1, and depend on the texture format.
- src1 must be a [Sampler \(Direct3D 9 asm-ps\)](#) (s#). src1 may not use a negate modifier (see [Destination Register Write Mask](#)). src1 may use swizzle (see [Source Register Swizzling](#)), which is applied after sampling, but before the write mask (see Destination Register Write Mask) is honored. The sampler must have been declared (using [dcl_samplerType \(sm2, sm3 - ps asm\)](#)) at the beginning of the shader.
- The number of coordinates required to perform the texture sample depends on how the sampler was declared. If it was declared as a cube, a three-component texture coordinate is required (.rgb). Validation enforces that coordinates provided to tex_ldl are sufficient for the texture dimension declared for the sampler. However, it is not guaranteed that the application actually sets a texture (through the API) with dimensions equal to the dimension declared for the sampler. In such a case, the runtime will attempt to detect mismatches (possibly in debug only). Sampling a texture with fewer dimensions than are present in the texture coordinate will be allowed and assumed to ignore the extra texture coordinate components. Conversely, sampling a texture with more dimensions than are present in the texture coordinate is illegal.

- If the src0 (texture coordinate) is [Temporary Register](#), the components required for the lookup (described above) must have been previously written.
- Sampling unsigned RGB textures will result in float values between 0.0 and 1.0.
- Sampling signed textures will result in float values between -1.0 to 1.0.
- When sampling floating-point textures, Float16 means that the data will range within MAX_FLOAT16. Float32 means the maximum range of the pipeline will be used. Sampling outside of either range is undefined.
- There is no dependent read limit.

Related topics

[Pixel Shader Instructions](#)

texldp - ps

Article • 08/19/2020 • 2 minutes to read

Projected texture load instruction. This instruction divides the input texture coordinate by the fourth element (.a or .w) just before sampling.

Syntax

```
texldp dst, src0, src1
```

where

- dst is the destination register.
- src0 is a source register that provides the texture coordinates for the texture sample. See [Texture Coordinate Register](#).
- src1 identifies the [Sampler \(Direct3D 9 asm-ps\)](#) (s#), where # specifies which texture sampler number to sample. The sampler has associated with it a texture and a sampler state defined by [D3DSAMPLERSTATETYPE](#).

For the set of restrictions when using texldp, see [texld](#).

Remarks

texldp performs projection on the coordinates read from src0 before performing the sample. Each texture coordinate is divided by src0.w, then the texture is sampled. When texldp completes, the contents of src0 are unaffected (unless dst is the same register). An alternative to using texldp is to manually perform the projection division in the shader. However, performing the division in shader code is usually slower than when performed by the texldp instruction, so avoid such additional math when possible.

The number of coordinates required for src0 to perform the texture sample depends on how src1 was declared, plus the .w component. Sampler types are declared with [dcl_samplerType \(sm2, sm3 - ps asm\)](#). If src1 is declared as a 2D sampler, then src0 must contain .xyw coordinates; if src1 is declared as either a cube sampler or a volume sampler, then src0 must contain .xyzw coordinates. Sampling a 2D texture with .xyzw coordinates is allowed (the .z coordinate is ignored).

If the source texture contains fewer than four components, defaults are placed in the missing components. Defaults depend on the texture format as shown in the following

table.

Texture Format	Default Values for missing components
D3DFMT_R5G6B5, D3DFMT_R8G8B8, D3DFMT_L8, D3DFMT_L16, D3DFMT_R3G3B2, D3DFMT_CxV8U8, D3DFMT_L6V5U5	A = 1.0
D3DFMT_V8U8, D3DFMT_V16U16, D3DFMT_G16R16, D3DFMT_G16R16F, D3DFMT_G32R32F	B = A = 1.0
D3DFMT_A8	R = G = B = 0.0
D3DFMT_R16F, D3DFMT_R32F	G = B = A = 1.0
All depth/stencil formats	R = B = 0.0, A = 1.0

This instruction is supported in the following versions:

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
texldp					x	x	x	x	x

ps_2_0 and ps_2_x

dst must be a [Temporary Register](#) (r#) and only .xyzw mask (default mask) is allowed.

src0 must be either a [Texture Coordinate Register](#) (t#) or a [Temporary Register](#) (r#), with no modifier or swizzle.

src1 must be a [Sampler \(Direct3D 9 asm-ps\)](#) (s#), with no modifier or swizzle.

If the D3DD3DPSHADERCAPS2_0_NODEPENDENTREADLIMIT cap bit is not set (in D3DPSHADERCAPS2_0), a given texture instruction ([texld](#), [texldp](#), [texldb - ps](#), [texldd](#)) may be dependent upon, at most, third order. A first-order dependent texture instruction is a texture instruction in which either:

- src0 is a [Temporary Register](#) (r#)
- dst was previously written, now being written again.

A second-order dependent texture instruction is defined as a texture instruction that reads or writes to a [Temporary Register](#) (r#) whose contents, before executing the texture instruction, depend (perhaps indirectly) on the outcome of a first-order

dependent texture instruction. An $(n)^{th}$ -order dependent texture instruction derives from an $(n - 1)^{th}$ -order texture instruction.

ps_3_0

For ps_3_0, src1 must be a [Sampler \(Direct3D 9 asm-ps\)](#) (s#), with no modifier. Swizzle is allowed on src1, and when applied, the texture lookup results are pre-swizzled before written to dst.

Related topics

[Pixel Shader Instructions](#)

texm3x2depth - ps

Article • 11/20/2019 • 2 minutes to read

Calculate the depth value to be used in depth testing for this pixel.

Syntax

```
texm3x2depth dst, src
```

where

- dst is the destination register.
- src is a source register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
texm3x2depth					x				

This instruction must be used with the [texm3x2pad - ps](#) instruction.

When using these two instructions, texture registers must use the following sequence.

```
tex t(n)                      // Define tn as a standard 3-vector.(tn must be
                               // defined in some way before it is used
texm3x2pad    t(m),   t(n)    // Where m > n
                               // Calculate z value
texm3x2depth t(m+1), t(n)    // Calculate w value; use both z and w to
                               // find depth
```

The depth calculation is done after using a dot product operation to find z and w. Here is more detail about how the depth calculation is accomplished.

The texm3x2pad instruction calculates z.

$$z = \text{TextureCoordinates(stage m)}_{UVW} * t(n)_{RGB}$$

The `texm3x2depth` instruction calculates w.

$$w = \text{TextureCoordinates(stage m+1)}_{UVW} * t(n)_{RGB}$$

Calculate depth and store the result in $t(m+1)$.

```
if (w == 0)
    t(m+1) = 1.0
else
    t(m+1) = z/w
```

The calculated depth is tagged to be used in the depth test for the pixel, replacing the existing depth test value for the pixel.

Be sure to clamp z/w to be in the range of (0-1). If z/w is outside this range, the result stored in the depth buffer will be undefined.

After running `texm3x2depth`, register $t(m+1)$ is no longer available for use in the shader.

When multisampling, using this instruction eliminates most of the benefit of the higher resolution depth buffer. Because the pixel shader runs once per pixel, the single depth value output by `texm3x2depth` or `texdepth - ps` will be used for each of the subpixel depth comparison tests.

Related topics

[Pixel Shader Instructions](#)

texm3x2pad - ps

Article • 11/20/2019 • 2 minutes to read

Performs the first row multiplication of a two-row matrix multiply. This instruction must be combined with either [texm3x2tex - ps](#) or [texm3x2depth - ps](#). Refer to either of these instructions for details on using texmpad.

Syntax

```
tex3x2pad dst, src
```

where

- dst is the destination register.
- src is a source register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
texm3x2pad	x	x	x						

This instruction cannot be used by itself.

Related topics

[Pixel Shader Instructions](#)

texm3x2tex - ps

Article • 11/20/2019 • 2 minutes to read

Performs the final row of a 3x2 matrix multiply and uses the result to do a texture lookup. texm3x2tex must be used in conjunction with the [texm3x2pad - ps](#) instruction.

Syntax

```
texm3x2tex dst, src
```

where

- dst is the destination register.
- src is a source register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
texm3x2tex	x	x	x						

The instruction is used as one of two instructions representing a 3x2 matrix multiply operation. This instruction must be used with the [texm3x2pad - ps](#) instruction.

When using these two instructions, texture registers must use the following sequence.

```
tex t(n)                                // Define tn as a standard 3-vector (tn must
                                          // be defined in some way before it is used)
texm3x2pad  t(m),  t(n)                  // where m > n
                                              // Perform first row of matrix multiply
texm3x2tex  t(m+1), t(n)                // Perform second row of matrix multiply
                                              // to get (u,v) to sample texture
                                              // associated with stage m+1
```

Here is more detail about how the 3x2 multiply is accomplished.

The `texm3x2pad` instruction performs the first row of the multiply to find u' .

$$u' = t(n)_{\text{RGB}} * \text{TextureCoordinates(stage } m\text{)}_{\text{uvw}}$$

The `texm3x2tex` instruction performs the second row of the multiply to find v' .

$$v' = t(n)_{\text{RGB}} * \text{TextureCoordinates(stage } m+1\text{)}_{\text{uvw}}$$

The `texm3x2tex` instruction samples the texture on stage $(m+1)$ with (u', v') and stores the result in $t(m+1)$.

$$t(m+1)_{\text{RGB}} = \text{TextureSample(stage } m+1\text{)}_{\text{RGB}} \text{ using } (u', v') \text{ as coordinates}$$

Examples

Here is an example shader with the texture maps and the texture stages identified.

```
ps_1_1
tex t0           // Bind texture in stage 0 to register t0
texm3x2pad t1, t0 // First row of matrix multiply
texm3x2tex t2, t0 // Second row of matrix multiply to get (u,v)
                   // with which to sample texture in stage 2
mov r0, t2       // Output result
```

This example requires the following textures in the following texture stages.

- Stage 0 takes a map with (x,y,z) perturbation data.
- Stage 1 holds texture coordinates. No texture is required in the texture stage.
- Stage 2 holds both texture coordinates as well as a 2D texture set at that texture stage.

Related topics

[Pixel Shader Instructions](#)

texm3x3 - ps

Article • 11/20/2019 • 2 minutes to read

Performs a 3x3 matrix multiply when used in conjunction with two [texm3x3pad - ps](#) instructions.

Syntax

```
texm3x3 dst, src
```

where

- dst is the destination register.
- src is a source register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
texm3x3		x	x						

This instruction is the same as the [texm3x3tex - ps](#) instruction, without the texture lookup.

This instruction is used as the final of three instructions representing a 3x3 matrix multiply operation. The 3x3 matrix is comprised of the texture coordinates of the third texture stage, and by the two preceding texture stages. Any texture assigned to any of the three texture stages is ignored.

This instruction must be used with two texm3x3pad instructions. Texture registers must follow the following sequence.

```
tex t(n)          // Define tn as a standard 3-vector (tn must
                  // be defined in some way before it is used)
texm3x3pad t(m), t(n) // where m > n
                      // Perform first row of matrix multiply
```

```
texm3x3pad t(m+1), t(n) // Perform second row of matrix multiply  
texm3x3    t(m+2), t(n) // Perform third row of matrix multiply to get a  
                      // 3-vector result
```

Here is more detail about how the 3x3 multiply is accomplished.

The first texm3x3pad instruction performs the first row of the multiply to find u' .

$$u' = \text{TextureCoordinates(stage m)}_{UVW} * t(n)_{RGB}$$

The second texm3x3pad instruction performs the second row of the multiply to find v' .

$$v' = \text{TextureCoordinates(stage m+1)}_{UVW} * t(n)_{RGB}$$

The texm3x3tex instruction performs the third row of the multiply to find w' .

$$w' = \text{TextureCoordinates(stage m+2)}_{UVW} * t(n)_{RGB}$$

Place the result of the matrix multiply in the destination register.

$$t(m+2)_{RGBA} = (u', v', w', 1)$$

Related topics

[Pixel Shader Instructions](#)

texm3x3pad - ps

Article • 11/20/2019 • 2 minutes to read

Performs the first or second row multiply of a three-row matrix multiply. This instruction must be used in combination with [texm3x3 - ps](#), [texm3x3spec - ps](#), [texm3x3vspec - ps](#), or [texm3x3tex - ps](#).

Syntax

```
texm3x3pad dst, src
```

where

- dst is the destination register.
- src is a source register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
texm3x3pad	x	x	x						

This instruction cannot be used by itself.

Related topics

[Pixel Shader Instructions](#)

texm3x3spec - ps

Article • 11/20/2019 • 3 minutes to read

Performs a 3x3 matrix multiply and uses the result to perform a texture lookup. This can be used for specular reflection and environment mapping. texm3x3spec must be used in conjunction with two [texm3x3pad - ps](#) instructions.

Syntax

```
texm3x3spec dst, src0, src1
```

where

- dst is the destination register.
- src0 and src1 are source registers.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
texm3x3spec	x	x	x						

This instruction performs the final row of a 3x3 matrix multiply, uses the resulting vector as a normal vector to reflect an eye-ray vector, and then uses the reflected vector to perform a texture lookup. The shader reads the eye-ray vector from a constant register. The 3x3 matrix multiply is typically useful for orienting a normal vector to the correct tangent space for the surface being rendered.

The 3x3 matrix is comprised of the texture coordinates of the third texture stage and the two preceding texture stages. The resulting post reflection vector (u,v,w) is used to sample the texture on the final texture stage. Any texture assigned to the preceding two texture stages is ignored.

This instruction must be used with two texm3x3pad instructions. Texture registers must use the following sequence.

```

tex t(n)                      // Define tn as a standard 3-vector (tn must
                                // be defined in some way before it is used)
texm3x3pad t(m),   t(n)        // where m > n
                                // Perform first row of matrix multiply
texm3x3pad  t(m+1), t(n)      // Perform second row of matrix multiply
texm3x3spec t(m+2), t(n), c0  // Perform third row of matrix multiply
                                // Then do a texture lookup on the texture
                                // associated with texture stage m+2

```

The first texm3x3pad instruction performs the first row of the multiply to find u' .

$$u' = \text{TextureCoordinates(stage } m\text{)}_{UVW} * t(n)_{RGB}$$

The second texm3x3pad instruction performs the second row of the multiply to find v' .

$$v' = \text{TextureCoordinates(stage } m+1\text{)}_{UVW} * t(n)_{RGB}$$

The texm3x3spec instruction performs the third row of the multiply to find w' .

$$w' = \text{TextureCoordinates(stage } m+2\text{)}_{UVW} * t(n)_{RGB}$$

The texm3x3spec instruction then does a reflection calculation.

$$(u', v', w') = 2 * [(N \cdot E) / (N \cdot N)] * N - E$$

// where the normal N is given by

$$// N = (u', v', w')$$

// and the eye-ray vector E is given by the constant register

// E = c# (Any constant register--c0, c1, c2, etc.--can be used)

Lastly, the texm3x3spec instruction samples $t(m+2)$ with (u', v', w') and stores the result in $t(m+2)$.

$t(m+2)_{RGBA} = \text{TextureSample(stage } m+2\text{)}_{RGBA}$ using (u', v', w') as coordinates

Examples

Here is an example shader with the texture maps and the texture stages identified.

```

ps_1_1
tex t0                      // Bind texture in stage 0 to register t0 (tn must
                                // be defined in some way before it is used)
texm3x3pad  t1,  t0          // First row of matrix multiply

```

```
texm3x3pad t2, t0      // Second row of matrix multiply
texm3x3spec t3, t0, c# // Third row of matrix multiply to get a 3-vector
                        // Reflect 3-vector by the eye-ray vector in c#
                        // Use reflected vector to lookup texture in
                        // stage 3
mov r0, t3             // Output the result
```

This example requires the following texture stage setup.

- Stage 0 is assigned a texture map with normal data. This is often referred to as a bump map. The data is (XYZ) normals for each texel. Texture coordinates at stage n defines where to sample this normal map.
- Texture coordinate set m is assigned to row 1 of the 3x3 matrix. Any texture assigned to stage m is ignored.
- Texture coordinate set m+1 is assigned to row 2 of the 3x3 matrix. Any texture assigned to stage m+1 is ignored.
- Texture coordinate set m+2 is assigned to row 3 of the 3x3 matrix. Stage m+2 is assigned a volume or cube texture map. The texture provides color data (RGBA).
- The eye-ray vector E is given by a constant register E = c#.

Related topics

[Pixel Shader Instructions](#)

texm3x3tex - ps

Article • 11/20/2019 • 2 minutes to read

Performs a 3x3 matrix multiply and uses the result to do a texture lookup. texm3x3tex must be used with two [texm3x3pad - ps](#) instructions.

Syntax

```
texm3x3tex dst, src
```

where

- dst is the destination register.
- src is a source register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
texm3x3tex	x	x	x						

This instruction is used as the final of three instructions representing a 3x3 matrix multiply operation, followed by a texture lookup. The 3x3 matrix is comprised of the texture coordinates of the third texture stage and the two preceding texture stages. The resulting three-component vector (u,v,w) is used to sample the texture in stage 3. Any texture assigned to the preceding two texture stages is ignored. The 3x3 matrix multiply is typically useful for orienting a normal vector to the correct tangent space for the surface being rendered.

This instruction must be used with the texm3x3pad instruction. Texture registers must use the following sequence.

```
tex t(n)          // Define tn as a standard 3-vector (tn must
                  // be defined in some way before it is used)
texm3x3pad t(m), t(n) // where m > n
```

```

        // Perform first row of matrix multiply
texm3x3pad t(m+1), t(n) // Perform second row of matrix multiply
texm3x3tex t(m+2), t(n) // Perform third row of matrix multiply to get a
                         // 3-vector with which to sample texture
                         // associated with texture stage m+2

```

Here is more detail about how the 3x3 multiply is accomplished.

The first texm3x3pad instruction performs the first row of the multiply to find u' .

$$u' = \text{TextureCoordinates(stage m)}_{UVW} * t(n)_{RGB}$$

The second texm3x3pad instruction performs the second row of the multiply to find v' .

$$v' = \text{TextureCoordinates(stage m+1)}_{UVW} * t(n)_{RGB}$$

The texm3x3spec instruction performs the third row of the multiply to find w' .

$$w' = \text{TextureCoordinates(stage m+2)}_{UVW} * t(n)_{RGB}$$

Lastly, the texm3x3tex instruction samples $t(m+2)$ with (u', v', w') and stores the result in $t(m+2)$.

$t(m+2)_{RGBA} = \text{TextureSample(stage m+2)}_{RGBA}$ using (u', v', w') as coordinates.

Examples

Here is an example shader with the texture maps identified and the texture stages identified.

```

ps_1_1
tex t0           // Bind texture in stage 0 to register t0
texm3x3pad t1, t0 // First row of matrix multiply
texm3x3pad t2, t0 // Second row of matrix multiply
texm3x3tex t3, t0 // Third row of matrix multiply to get a
                   // 3-vector with which to sample texture at
                   // stage 3 output result
mov r0, t3       // stage 3 output result

```

This example requires the following texture stage setup.

- Stage 0 is assigned a texture map with normal data. This is often referred to as a bump map. The data is (XYZ) normals for each texel. Texture coordinate set 0 defines how to sample this normal map.
- Texture coordinate set 1 is assigned to row 1 of the 3x3 matrix. Any texture assigned to stage 1 is ignored.

- Texture coordinate set 2 is assigned to row 2 of the 3x3 matrix. Any texture assigned to stage 2 is ignored.
- Texture coordinate set 3 is assigned to row 3 of the 3x3 matrix. A volume or cube texture should be set to stage 3 for lookup by the transformed 3D vector.

Related topics

[Pixel Shader Instructions](#)

texm3x3vspec - ps

Article • 11/20/2019 • 3 minutes to read

Performs a 3x3 matrix multiply and uses the result to perform a texture lookup. This can be used for specular reflection and environment mapping where the eye-ray vector is not constant. `texm3x3vspec` must be used in conjunction with two [texm3x3pad - ps](#) instructions. If the eye-ray vector is constant, the [texm3x3spec - ps](#) instruction will perform the same matrix multiply and texture lookup.

Syntax

```
texm3x3vspec dst, src
```

where

- `dst` is the destination register.
- `src` is a source register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
texm3x3vspec	x	x	x						

This instruction performs the final row of a 3x3 matrix multiply operation, interprets the resulting vector as a normal vector to reflect an eye-ray vector, and then uses the reflected vector as a texture address for a texture lookup. It works just like [texm3x3spec - ps](#), except that the eye-ray vector is taken from the fourth component of the texture coordinates. The 3x3 matrix multiply is typically useful for orienting a normal vector to the correct tangent space for the surface being rendered.

The 3x3 matrix is comprised of the texture coordinates of the third texture stage and the two preceding texture stages. The resulting post-reflection vector (UVW) is used to sample the texture in stage 3. Any texture assigned to the preceding two texture stages is ignored.

This instruction must be used with the texm3x3pad instruction. Texture registers must use the following sequence.

```

tex t(n)           // Define tn as a standard 3-vector (tn must
                   // be defined in some way before it is used)
texm3x3pad t(m), t(n) // where m > n
                      // Perform first row of matrix multiply
texm3x3pad t(m+1), t(n) // Perform second row of matrix multiply
texm3x3vspec t(m+2), t(n) // Perform third row of matrix multiply
                           // Then do a texture lookup on the texture
                           // associated with texture stage m+2

```

The first texm3x3pad instruction performs the first row of the multiply to find u' .

$$u' = \text{TextureCoordinates(stage m)}_{UVW} * t(n)_{RGB}$$

The second texm3x3pad instruction performs the second row of the multiply to find v' .

$$v' = \text{TextureCoordinates(stage m+1)}_{UVW} * t(n)_{RGB}$$

The texm3x3spec instruction performs the third row of the multiply to find w' .

$$w' = \text{TextureCoordinates(stage m+2)}_{UVW} * t(n)_{RGB}$$

The texm3x3vspec instruction also does a reflection calculation.

$$(u', v', w') = 2 * [(N * E) / (N * N)] * N - E$$

// where the normal N is given by

$$// N = (u', v', w')$$

// and the eye-ray vector E is given by

$$// E = (\text{TextureCoordinates(stage m)}_Q,$$

$$// \text{TextureCoordinates(stage m+1)}_Q,$$

$$// \text{TextureCoordinates(stage m+2)}_Q)$$

Lastly, the texm3x3vspec instruction samples $t(m+2)$ with (u', v', w') and stores the result in $t(m+2)$.

$$t(m+2)_{RGB} = \text{TextureSample(stage m+2)}_{RGB} \text{ using } (u', v', w') \text{ as coordinates}$$

Examples

Here is an example shader with the texture maps identified and the texture stages identified.

```
ps_1_1
tex t0           // Bind texture in stage 0 to register t0
texm3x3pad    t1,  t0 // First row of matrix multiply
texm3x3pad    t2,  t0 // Second row of matrix multiply
texm3x3vspec  t3,  t0 // Third row of matrix multiply to get a 3-vector
                      // Reflect 3-vector by the eye-ray vector
                      // Use reflected vector to do a texture lookup
                      // at stage 3
mov r0, t3       // Output the result
```

This example requires the following texture stage setup.

- Stage 0 is assigned a texture map with normal data. This is often referred to as a bump map. The data is (XYZ) normals for each texel. Texture coordinates at stage n defines how to sample this normal map.
- Texture coordinate set m is assigned to row 1 of the 3x3 matrix. Any texture assigned to stage m is ignored.
- Texture coordinate set m+1 is assigned to row 2 of the 3x3 matrix. Any texture assigned to stage m+1 is ignored.
- Texture coordinate set m+2 is assigned to row 3 of the 3x3 matrix. Stage m+2 is assigned a volume or cube texture map. The texture provides color data (RGBA).
- The eye-ray vector E is passed into the instruction in the fourth component (q) of the texture coordinate data at stages m, m+1, and m+2.

Related topics

[Pixel Shader Instructions](#)

texreg2ar - ps

Article • 08/19/2020 • 2 minutes to read

Interprets the alpha and red color components of the source register as texture address data (u,v) to sample the texture at the stage corresponding to the destination register number. The result is stored in the destination register.

Syntax

```
texreg2ar dst, src
```

where

- dst is the destination register.
- src is a source register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
texreg2ar	x	x	x						

This instruction is useful for color-space remapping operations.

Here is an example of the sequence the instruction follows:

```
tex t(n) texreg2ar t(m), t(n) where m > n // The first instruction loads the texture color  
(RGBA) // into register tn tex tn // The second instruction remaps the color t(m)RGBA =  
TextureSample(stage m)RGBA using t(n)AR as coordinates
```

_bx2 cannot be used on the src register for texreg2ar or [texreg2gb - ps](#) instructions.

For this instruction, the source register must use unsigned data. Use of signed or mixed data in the source register will produce undefined results. For more information, see [D3DFORMAT](#).

Related topics

Pixel Shader Instructions

texreg2gb - ps

Article • 08/19/2020 • 2 minutes to read

Interprets the green and blue color components of the source register as texture address data to sample the texture at the stage corresponding to the destination register number.

Syntax

```
texreg2gb dst, src
```

where

- dst is the destination register.
- src is a source register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
texreg2gb		x	x						

This instruction is useful for color-space remapping operations.

Here is an example of the sequence the instruction follows.

```
tex t(n) texreg2gb t(m), t(n) where m > n // The first instruction loads the texture color  
(RGBA) // into register tn tex tn // The second instruction remaps the color t(m)RGBA =  
TextureSample(stage m)RGBA using t(n)GB as coordinates
```

_bx2 cannot be used on the src register for [texreg2ar - ps](#) or texreg2gb instructions.

For this instruction, the source register must use unsigned data. Use of signed or mixed data in the source register will produce undefined results. For more information, see [D3DFORMAT](#).

Related topics

Pixel Shader Instructions

texreg2rgb - ps

Article • 11/20/2019 • 2 minutes to read

Interprets the red, green, and blue (RGB) color components of the source register as texture address data in order to sample the texture at the stage corresponding to the destination register number. The result is stored in the destination register.

Syntax

```
texreg2rgb dst, src
```

where

- dst is the destination register.
- src is a source register.

Remarks

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
texreg2rgb		x	x						

This instruction is useful for color-space remapping operations. It supports two-dimensional (2D) and three-dimensional (3D) coordinates. It can be used just like the [texreg2ar - ps](#) or [texreg2gb - ps](#) to remap 2D data. However, this instruction also supports 3D data so it can be used with cube maps and 3D volume textures.

Here is an example of the sequence the instruction follows.

```
tex t(n)
texreg2rgb t(m), t(n)      where m > n
```

Here is more detail about how the remapping is accomplished.

// The first instruction loads the texture color (RGBA) into register tn
tex tn // The second instruction remaps the color t(m)_{RGBA} = TextureSample(stage m)_{RGBA} using

$t(n)_{RGB}$ as coordinates

Related topics

[Pixel Shader Instructions](#)

Registers

Article • 08/23/2019 • 2 minutes to read

- [ps_1_1_ps_1_2_ps_1_3_ps_1_4 Registers](#)
- [ps_2_0 Registers](#)
- [ps_2_x Registers](#)
- [ps_3_0 Registers](#)

Related topics

[Pixel Shaders](#)

Pixel Shader Register Modifiers

Article • 08/23/2019 • 2 minutes to read

Source Register Modifiers:

- [Source Register Bias](#)
- [Source Register Invert](#)
- [Source Register Negate](#)
- [Source Register Swizzling](#)
- [Source Register Scale x 2](#)
- [Source Register Signed Scaling](#)
- [Destination Register Write Mask](#)

Destination Register Modifiers:

- [Destination Register Write Mask](#)

Related topics

[Registers](#)

Pixel Shader Source Register Modifiers

Article • 06/30/2021 • 2 minutes to read

Use source register modifiers to change the value read from a register before an instruction runs. The contents of a source register are left unchanged. Modifiers are useful for adjusting the range of register data in preparation for the instruction. A set of modifiers called selectors copies or replicates the data from a single channel (r,g,b,a) into the other channels.

ps_1_1 - ps_1_4

This table identifies the versions that support each modifier:

Source register modifiers	Syntax	Version 1_1	Version 1_2	Version 1_3	Version 1_4
bias	register_bias	X	X	X	X
invert	1 - register	X	X	X	X
negate	- register	X	X	X	X
scale by 2	register_x2				X
signed scaling	register_bx2	X	X	X	X
texld and texcrd modifiers	register_d*	X	X	X	X
source register swizzling	register.xyzw	X	X	X	X

Source register modifiers can be used only on arithmetic instructions. They cannot be used on texture address instructions. The exception to this is the [scale by 2](#) modifier. For version 1_1, signed scale can be used on the source argument of any texm* instruction. For version 1_2 or 1_3, signed scale can be used on the source argument of any texture address instruction.

Some modifier specific restrictions:

- Negate can be combined with either the bias, signed scaling, or scalex2 modifier.
When combined, negate is run last.
- Invert cannot be combined with any other modifier.

- Invert, negate, bias, signed scaling, and scalex2 can be combined with any of the selectors.
- Source register modifiers should not be used on constant registers because they will cause undefined results. For version 1_4, modifiers on constants are not allowed and will fail validation.

ps_2_0 and Above

For version ps_2_0 and up, the number of modifiers has been simplified.

Negate

Negate the contents of the source register.

Component modifier	Description
- r	Source negation

The negate modifier cannot be used on second source register of these instructions: [m3x2 - ps](#), [m3x3 - ps](#), [m3x4 - ps](#), [m4x3 - ps](#), and [m4x4 - ps](#).

Pixel shader versions	2_0	2_x	2_sw	3_0	3_sw
-	x	x	x	x	x

Absolute Value

Take the absolute value of the register.

Pixel shader versions	2_0	2_x	2_sw	3_0	3_sw
abs				x	x

If any version 3 shader reads from one or more constant float registers (c#), one of the following must be true.

- All of the constant floating-point registers must use the abs modifier.

- None of the constant floating-point registers can use the abs modifier.

Related topics

[Pixel Shader Register Modifiers](#)

Source Register Bias

Article • 08/23/2019 • 2 minutes to read

Subtract 0.5 from all components.

Registers

Source register. For more about register types, see [ps_1_1_ps_1_2_ps_1_3_ps_1_4 Registers](#).

Remarks

The contents of the register are not changed. The modifier is applied only to the data read from the register. The bias is applied to all four color channels (RGBA) as follows:

```
output = (input - 0.5)
```

The effect is to modify data that was in the range 0 to 1 to be in the range -0.5 to 0.5. Applying bias to data outside this range may produce undefined results.

ⓘ Note

This modifier is mutually exclusive with [Source Register Invert](#), so it cannot be applied to the same register.

This modifier is for use with the arithmetic instructions.

Example

This example performs the same operation as D3DTOP_ADDSIGNED in DirectX 6.0 and 7.0 multiple texture syntax.

```
add r0, r0, t0_bias; Shift down by 0.5.
```

Related topics

[Pixel Shader Source Register Modifiers](#)

Source Register Invert

Article • 08/23/2019 • 2 minutes to read

Performs a $(1 - \text{value})$ calculation for each channel of the specified register.

Syntax

```
1 - register
```

Registers

Source Register. For more about register types, see [ps_1_1_ps_1_2_ps_1_3_ps_1_4 Registers](#).

Remarks

The contents of the register are not changed. The modifier is applied only to the data read from the register. The invert operation is applied to all four color channels (RGBA).

This modifier can be used only with arithmetic instructions. In addition, this modifier cannot be combined with the other [Destination Register Write Mask](#).

Example

This example uses inversion to generate the complement of register r1.

```
mul r0, r0, 1-r1;
```

Related topics

[Pixel Shader Source Register Modifiers](#)

Source Register Negate

Article • 08/23/2019 • 2 minutes to read

Performs a negate ($y = -x$), on all register components.

Syntax

```
- register
```

Registers

Source Register. For more about register types, see [ps_1_1_ps_1_2_ps_1_3_ps_1_4 Registers](#).

Remarks

The contents of the register are not changed. The modifier is applied only to the data read from the register. The negate operation is applied to all four color channels (RGBA).

This operation is performed after any other modifiers present on the same argument.

This modifier is mutually exclusive with [Source Register Invert](#) so it cannot be applied to the same register.

This modifier is for use only with arithmetic instructions.

Example

The following example shows how to use this modifier.

```
mul r0, r0, -v1;
```

Related topics

[Pixel Shader Source Register Modifiers](#)

Source Register Scale x 2

Article • 08/23/2019 • 2 minutes to read

Multiply the value by two before using it in the instruction.

Syntax

```
register_x2
```

Register

Source register. For more about register types, see [ps_1_1_ps_1_2_ps_1_3_ps_1_4 Registers](#).

Remarks

The contents of the register are not changed. The modifier is applied only to the data read from the register.

This modifier is mutually exclusive with [Source Register Invert](#), so it cannot be applied to the same register.

This modifier is only available to arithmetic instructions in version 1_4.

Example

This example samples a texture, converts data to the range of -1 to +1, and calculates a dot product.

```
mul r0, r0, r1_x2;
```

Related topics

[Pixel Shader Source Register Modifiers](#)

Source Register Signed Scaling

Article • 08/23/2019 • 2 minutes to read

Subtracts 0.5 from each channel and scales the result by 2.0. The name bx2 comes from bias and scale-times-two, which is the operation it performs.

Syntax

```
source register_bx2
```

Register

Source Register. For more about register types, see [ps_1_1_ps_1_2_ps_1_3_ps_1_4 Registers](#).

Remarks

This operation is commonly used to expand data from [0.0 to 1.0] to [-1.0 to 1.0]. This modifier is designed for use with the arithmetic instructions. This modifier is commonly used on inputs to the dot product instruction ([dp3 - ps](#)). Using _bx2 on data outside the range 0 to 1 may produce undefined results.

The signed scaling operation is applied to the data read from the register before the next instruction is run. The operation is applied to all four color channels (RGBA) as follows:

$$y = 2(x - 0.5)$$

The contents of the register are not changed. The modifier is applied only to the data read from the register.

This modifier is mutually exclusive with [Source Register Invert](#) so it cannot be applied to the same register.

Version information:

- For ps_1_0 and ps_1_1, you can use _bx2 on any source register for texture instructions of the form texm3x2* and texm3x3*. _bx2 cannot be used on any of the other texture instructions such as [texreg2ar - ps](#) or [texreg2gb - ps](#).
- For ps_1_2 and ps_1_3, you can use _bx2 on any source register for any tex* instruction except: [texreg2ar - ps](#), [texreg2gb - ps](#), [texbem - ps](#) or [texbeml - ps](#).

Example

This example samples a texture, converts data to the range of -1 to +1, and calculates a dot product.

```
tex t0          ; Read a texture color.  
dp3_sat r0, t0_bx2, v0_bx2 ; Calculate a dot product.
```

Related topics

[Pixel Shader Source Register Modifiers](#)

ps_1_4 source register modifiers for texId, texcrd

Article • 11/27/2019 • 2 minutes to read

Two pixel shader version 1_4 texture address instructions, [texId - ps_1_4](#) and [texcrd - ps](#), have custom syntax. These instructions support their own set of source register modifiers, source register selectors, and destination-register write masks, as shown here.

Source Register Modifiers for texId and texcrd

These modifiers provide projective divide functionality by dividing the x and y values by either the z or w values.

Source register modifiers	Description	Syntax
_dz	Divide x,y components by z	register_dz
_db	Divide x,y components by z	register_db
_dw	Divide x,y components by w	register_dw
_da	Divide x,y components by w	register_da

Remarks

The _dz or _db modifier does the following:

```
x' = x/z ( x' = 1.0 if z == 0)
y' = y/z ( y' = 1.0 if z == 0)
z' is undefined
w' is undefined
```

The _dw or _da modifier does the following:

```
x' = x/w ( x' = 1.0 if w == 0)
y' = y/w ( y' = 1.0 if w == 0)
```

```
z' is undefined  
w' is undefined
```

Related topics

[Pixel Shader Source Register Modifiers](#)

Source register swizzling (HLSL PS reference)

Article • 11/23/2019 • 2 minutes to read

Swizzling refers to the ability to copy any source register component to any temporary register component. Swizzling does not affect the source register data. Before an instruction runs, the data in a source register is copied to a temporary register.

Source Swizzling

Source swizzle allows individual component of a source register to take on the value of any of the four components of the same source register before the register is read for computation.

For example, the `.zxy` swizzle means:

- `.x` component will take on the value of `.z` component
- `.y` component will take on the value of `.x` component
- `.z` component will take on the value of `.x` component
- `.w` component will take on the value of `.y` component

The components can appear in any order. If fewer than four components are specified, the last component is repeated. For example:

```
.xy  = .xyyy  
.wzx = .wzxx  
.z   = .zzzz
```

If no component is specified, no swizzling is applied.

Some instructions have restrictions for source swizzle. They are listed in the respected instruction reference pages.

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
<code>.x</code>				x	x	x	x	x	x
<code>.y</code>				x	x	x	x	x	x
<code>.z</code>	x*	x*	x*	x	x	x	x	x	x

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
.w	x	x	x	x	x	x	x	x	x
.xyzw (default)	x	x	x	x	x	x	x	x	x
.yzxw					x	x	x	x	x
.zxyw					x	x	x	x	x
.wzyx					x	x	x	x	x
arbitrary swizzle					x	x	x	x	x

* Only available if destination write mask is .w (.a).

Arbitrary Swizzle

Swizzles can be applied to source registers in an arbitrary order; that is, any source register can take any component mask, in any order.

Replicate Swizzle

Replicate swizzle copies one component to another. This is, exactly one of the .x, .y, .z, .w swizzle components (or the .r, .g, .b, .a equivalents) must be specified.

Related topics

[Pixel Shader Source Register Modifiers](#)

[ps_1_1_ps_1_2_ps_1_3_ps_1_4 Registers](#)

[ps_2_0 Registers](#)

[ps_2_x Registers](#)

[ps_3_0 Registers](#)

Destination Register Write Mask

Article • 08/23/2019 • 2 minutes to read

A write mask controls which components of a destination register are written after an instruction is completed. An output write mask is allowed as long as the components are in the order of .rgba or .xyzw. That is, .rba and .xw are valid masks. Texture registers have one set of rules and non-texture registers have another set of rules.

Syntax

```
dst.writemask
```

where

- dst is a destination register.
- writemask is a sequence of masks from either set: (x,y,z,w) or (red, green, blue, alpha).

Remarks

The following destination write masks are available.

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_x	2_sw	3_0	3_sw
.xyzw (default)	x	x	x	x	x	x	x	x	x
.xyz	x	x	x	x	x	x	x	x	x
.w	x	x	x	x	x	x	x	x	x
arbitrary mask				x	x	x	x	x	x

The arbitrary mask allows any set of channels to be combined to produce a mask. The channels must be listed in the order r, g, b, a - for example, register.rba, which updates the red, blue, and alpha channels of the destination. The arbitrary mask is available in version 1_4 or higher.

If no destination write mask is specified, the destination write mask defaults to the `rgba` case. In other words, all channels in the destination register are updated.

For versions 1_0 to 1_3, the `dp3 - ps` `dp3` arithmetic instruction can use only the `.rgb` or `.rgba` output write masks.

Destination register write masks are supported for arithmetic operations only. They cannot be used on texture addressing instructions, with the exception of the version 1_4 instructions, `texcrd - ps` and `texld - ps_2_0 and up`.

The default is to write all four color channels.

```
// All four color channels can be written by explicitly listing them.  
mul r0.rgb, t0, v0  
  
// Or, the default mask can be used to write all four channels.  
mul r0, t0, v0
```

The alpha write mask is also referred to as the scalar write mask, because it uses the scalar pipeline.

```
add r0.a, t1, v1
```

So this instruction effectively puts the sum of the alpha component of `t1` and the alpha component of `v1` into `r0.a`.

The color write mask is used to control writing to the color channels.

```
// The color write mask is also referred to as the vector write mask,  
// because it uses the vector pipeline.  
mul r0.rgb, t0, v0
```

For version 1_4, destination write masks can be used in any combination as long as the masks are ordered `r,g,b,a`.

```
// This example updates the red, blue, and alpha channels.  
mov r0.rba, r1
```

A co-issued instruction allows two potentially different instructions to be issued simultaneously. This is accomplished by executing the instructions in the alpha pipeline and the RGB pipeline.

```
mul r0.rgb, t0, v0  
+ add r1.a, t1, c1
```

The advantage of pairing instructions this way is that it allows different operations to be performed in the vector and scalar pipeline in parallel.

These vertex shader output registers are restricted to the following write masks:

Register Type	Required Write Mask
oFog	no explicit write mask is allowed on this scalar register
oPts	no explicit write mask is allowed on this scalar register
oPos	.xyzw(which is the default)
oT#	combined mask: .x .xy .xyz .xyzw (which is the default)

Related topics

[Pixel Shader Register Modifiers](#)

ps_1_1_ps_1_2_ps_1_3_ps_1_4 Registers

Article • 06/30/2021 • 5 minutes to read

Pixel shaders depend on registers to get vertex data, to output pixel data, to hold temporary results during calculations and to identify texture sampling stages. There are several types of registers, each with a unique functionality. This section contains reference information for the input and output registers implemented by pixel shader version 1_X.

Registers hold data for use by the pixel shader. Registers are fully described in the following sections.

- Register Types describes the four types of registers available and their purposes.
- Read Port Limit details the restrictions on using multiple registers in a single instruction.
- Read only Read Write describes which registers can be used for reading, writing, or both.
- Range details the range of the component data.

Register Types

Name	Type	Version 1_1	Version 1_2	Version 1_3	Version 1_4
c#	Constant register	8	8	8	8
r#	Temporary register	2	2	2	6
t#	Texture register	4	4	4	6
v#	Color register	2	2	2	2 in phase 2

- Constant registers contain constant data. Data can be loaded into a constant register using [SetPixelShaderConstantF](#) or it can be defined using [def - ps](#). Constant registers are not usable by texture address instructions. The only exception is the [texm3x3spec - ps](#) instruction, which uses a constant register to supply an eye-ray vector.
- Temporary registers are used to store intermediate results. r0 additionally serves as the pixel shader output. The value in r0 at the end of the shader is the pixel color for the shader.

Shader validation will fail [CreatePixelShader](#) on any shader that attempts to read from a temporary register that has not been written by a previous instruction.

[D3DXAssembleShader](#) will fail similarly, assuming validation is enabled (do not use D3DXSHADER_SKIPVALIDATION).

- Texture registers

For pixel shader version 1_1 to 1_3, texture registers contain texture data or texture coordinates. Texture data is loaded into a texture register when a texture is sampled. Texture sampling uses texture coordinates to look up, or sample, a color value at the specified (u,v,w,q) coordinates while taking into account the texture stage state attributes. The texture coordinate data is interpolated from the vertex texture coordinate data and is associated with a specific texture stage. There is a default one-to-one association between texture stage number and texture coordinate declaration order. By default, the first set of texture coordinates defined in the vertex format is associated with texture stage 0.

For these pixel shader versions, texture registers behave just like temporary registers when used by arithmetic instructions.

For pixel shader version 1_4, texture registers (t#) contain read-only texture coordinate data. This means that the texture coordinate set and the texture stage number are independent from each other. The texture stage number (from which to sample a texture) is determined by the destination register number (r0 to r5). For the texId instruction, the texture coordinate set is determined by the source register (t0 to t5), so the texture coordinate set can be mapped to any texture stage. In addition, the source register (specifying texture coordinates) for texId can also be a temporary register (r#), in which case the contents of the temporary register are used as texture coordinates.

- Color registers contain per-pixel color values. The values are obtained by per-pixel iteration of the diffuse and specular color values in the vertex data. For pixel shader version 1_4 shaders, color registers are available only during the second phase.

If the shade mode is set to D3DSHADE_FLAT, the iteration of both vertex colors (diffuse and specular) is disabled. Regardless of the shade mode, fog will still be iterated by the pipeline if pixel fog is enabled. Keep in mind that fog is applied later in the pipeline than the pixelshader.

It is common to load the v0 register with the vertex diffuse color data. It is also common to load the v1 register with the vertex specular color data.

Input color data values are clamped (saturated) to the range 0 through 1 because this is the valid input range for color registers in the pixel shader.

Pixel shaders have read only access to color registers. The contents of these registers are iterated values, but iteration may be performed at much lower precision than texture coordinates.

Read Port Limit

The read port limit specifies the number of different registers of each register type that can be used as a source register in a single instruction.

Name	Type	Version 1_1	Version 1_2	Version 1_3	Version 1_4
c#	Constant register	2	2	2	2
r#	Temporary register	2	2	2	3
t#	Texture register	2	3	3	1
v#	Color register	2	2	2	2 in phase 2

For example, the color registers for almost all versions have a read port limit of two. This means that a single instruction can use a maximum of two different color registers (v0 and v1 for instance) as source registers. This example shows two color registers being used in the same instruction:

```
mad r0, v1, c2, v0
```

Read-only, Read/Write

The register types are identified according to read-only (RO) capability or read/write (RW) capability in the following table. Read-only registers can be used only as source registers in an instruction; they can never be used as a destination register.

Name	Type	Version 1_1	Version 1_2	Version 1_3	Version 1_4
Name	Type	1_1	1_2	1_3	1_4
c#	Constant register	RO	RO	RO	RO

Name	Type	Version 1_1	Version 1_2	Version 1_3	Version 1_4
r#	Temporary register	RW	RW	RW	RW
t#	Texture register	RW	RW	RW	See following note
v#	Color register	RO	RO	RO	RO

Registers that are RW capable can be used to store intermediate results. This includes the temporary registers and texture registers for some of the shader versions.

① Note

- For pixel shader version 1_4, texture registers are RO for texture addressing instructions, and texture registers can be neither read from nor written to by arithmetic instructions. Also, because texture registers have become texture coordinate registers, having RO access is not a regression of previous functionality.

Range

The range is the maximum and minimum register data value. The ranges vary based on the type of register. The ranges for some of the registers can be queried from the device caps using [GetDeviceCaps](#).

Name	Type	Range	Versions
c#	Constant register	-1 to +1	All versions
r#	Temporary register	- PixelShader1x.MaxValue to + PixelShader1x.MaxValue	All versions
t#	Texture register	- MaxTextureRepeat to + MaxTextureRepeat	All versions
v#	Color register	0 to 1	All versions

Early pixel shader hardware represents data in registers using a fixed-point number. This limits precision to a maximum of approximately eight bits for the fractional part of a

number. Keep this in mind when designing a shader.

For pixel shader version 1_1 to 1_3, MaxTextureRepeat must be a minimum of one. For 1_4, MaxTextureRepeat must be a minimum of eight.

See [D3DCAPS9](#) for more information about PixelShader1xMaxValue.

Related topics

[Registers](#)

ps_2_0 Registers

Article • 06/18/2021 • 2 minutes to read

Pixel shaders depend on registers to get vertex data, to output pixel data, to hold temporary results during calculations, and to identify texture sampling stages. There are several types of registers, each with a unique functionality. This section contains reference information for the input and output registers implemented by pixel shader version 2_x.

Input Register Types

Register	Name	Count	R/W	# Read ports	# Reads/inst	Dimension	RelAddr	Defaults	Requires DCL
v#	Input Color Register	2	R	1	Unlimited	4	N	Partial(0001). See note 4	Y
r#	Temporary Register	See note 1	R/W	3	Unlimited	4	N	None	N
c#	Constant Float Register	32	R	1	2	4	N	0000	N
i#	Constant Integer Register	16	See note 2	1	1	4	N	0000	N
b#	Constant Boolean Register	16	See note 2	1	1	1	N	FALSE	N
p0	Predicate Register	1	See note 2	1	1	1	N	None	Y
s#	Sampler (Direct3D 9 asm-ps)	16	See note 3	1	1	4	N	See note 5	Y
t#	Texture Coordinate Register	8	R	1	1	4	N	None	Y

Notes:

1. 12 min/32 max: The number of r# registers is determined by D3DPSHADERCAPS2_0.NumTemps (which ranges from 12 to 32).
2. Only usable by a flow control instruction.
3. Only usable by a texture sampling instruction.
4. partial(x, y, z, w) - If only a subset of channels are updated in the register, the remaining channels will default to specified values (x, y, z, w).
5. Defaults for sampler lookups exist, but values depend on texture format.

The number of readports is the number of different registers (for each register type) that can be read in a single instruction.

Output Register Types

Register	Name	Count	R/W	Dimension	RelAddr	Defaults	Requires DCL
oC#	Output Color Register	See Multiple-element Textures (Direct3D 9)	W	4	N	None	N
oDepth	Output Depth Register	1	W	1	N	None	N

Related topics

[Registers](#)

ps_2_x Registers

Article • 06/18/2021 • 2 minutes to read

Pixel shaders depend on registers to get vertex data, to output pixel data, to hold temporary results during calculations, and to identify texture sampling stages. There are several types of registers, each with a unique functionality. This section contains reference information for the input and output registers implemented by pixel shader version 2_x.

Input Register Types

Register	Name	Count	R/W	# Read ports	# Reads/inst	Dimension	RelAddr	Defaults	Requires DCL
v#	Input Color Register	2	R	1	Unlimited	4	N	Partial(0001). See note 4	Y
r#	Temporary Register	See note 1	R/W	3	Unlimited	4	N	None	N
c#	Constant Float Register	32	R	1	2	4	N	0000	N
i#	Constant Integer Register	16	See note 2	1	1	4	N	0000	N
b#	Constant Boolean Register	16	See note 2	1	1	1	N	FALSE	N
p0	Predicate Register	1	See note 2	1	1	1	N	None	Y
s#	Sampler (Direct3D 9 asm-ps)	16	See note 3	1	1	4	N	See note 5	Y
t#	Texture Coordinate Register	8	R	1	1	4	N	None	Y

Notes:

1. 12 min/32 max: The number of r# registers is determined by D3DPSHADERCAPS2_0.NumTemps (which ranges from 12 to 32).
2. Only usable by a flow control instruction.
3. Only usable by a texture sampling instruction.
4. partial(x, y, z, w) - If only a subset of channels are updated in the register, the remaining channels will default to specified values (x, y, z, w).
5. Defaults for sampler lookups exist, but values depend on texture format.

The number of readports is the number of different registers (for each register type) that can be read in a single instruction.

Output Register Types

Register	Name	Count	R/W	Dimension	RelAddr	Defaults	Requires DCL
oC#	Output Color Register	See Multiple-element Textures (Direct3D 9)	W	4	N	None	N
oDepth	Output Depth Register	1	W	1	N	None	N

Related topics

[Registers](#)

ps_3_0 Registers

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

Pixel shaders depend on registers to get vertex data, to output pixel data, to hold temporary results during calculations, and to identify texture sampling stages. There are several types of registers, each with a unique functionality. This section contains reference information for the input and output registers implemented by pixel shader version 3_0.

New Registers

Input Register

The Input Registers (v#) are now fully floating point and the [Texture Coordinate Registers](#) (t#) have been consolidated into it. The [dcl_semantics \(sm3 - ps asm\)](#) at the top of the shader is used to describe what is contained in a particular Input_Register. A semantic for the pixel types is introduced (analogous to the vertex side) for this model. No clamping is performed when the input registers are defined as colors (like texture coordinates). The evaluation of the registers defined as color differs from the texture coordinates when multisampling.

Face Register

The face register (vFace) is new for this model. This is a floating point scalar register that will eventually contain the primitive area. In ps_3_0, however, only the sign of this register is valid. Hence, if the value is less than zero (the sign bit is set negative) the primitive is the back face (the area is negative, counterclockwise). Therefore, in ps_3_0 it only makes sense to compare this register against 0 (> 0 or < 0). Inside the pixel shader, the application can make a decision as to which lighting technique to use. Two-sided lighting can be achieved this way. This register requires a declaration, so undeclared usage will be flagged as an error. For lines and point primitives, this register is undefined. The face register can only be used as condition with the following instructions: [setp_comp - ps](#), [if_comp - ps](#), or [break_comp - ps](#).

Loop Counter Register

The [Loop Counter Register](#) (aL) is new for this model. It automatically gets incremented in each execution of the [loop - ps/endloop - ps](#) block. It can be used in the block for relative addressing if needed. It is invalid to use Loop Counter Register outside the loop.

Position Register

The Position Register (vPos) is new for this model. It contains the current pixels (x, y) in the corresponding channels. The (z, w) channels are undefined. This register requires a declaration, so undeclared usage will be flagged as an error. When declared, this register must have exactly one of the following masks: .x, .y, .xy.

Input Register Types

Register	Name	Count	R/W	#	#	Dimension	RelAddr	Defaults	Requires DCL
				Read ports	Reads/inst				
v#	Input Register	10	R	1	Unlimited	4	aL	None	Yes
r#	Temporary Register	32	R/W	3	Unlimited	4	No	None	No
c#	Constant Float Register	224	R	1	Unlimited	4	No	0000	No
i#	Constant Integer Register	16	R	1	1	4	No	0000	No
b#	Constant Boolean Register	16	R	1	1	1	No	FALSE	No
p0	Predicate Register	1	R	1	1	1	No	None	No
s#	Sampler (Direct3D 9 asm-ps)	16	R	1	1	4	No	See note 1	Yes
vFace	Face_Register	1	R	1	Unlimited	1	No	None	Yes
vPos	Position_Register	1	R	1	Unlimited	4	No	None	Yes
aL	Loop_Counter_Register	1	R	1	Unlimited	1	n/a	None	No

Notes:

- Defaults for sampler lookups exist, but values depend on texture format.

The number of readports is the number of different registers (for each register type) that can be read in a single instruction.

Output Register Types

Register	Name	Count	R/W	Dimension	RelAddr	Defaults	Requires DCL
oC#	Output Color Register	See Multiple-element Textures (Direct3D 9)	W	4	No	None	No
oDepth	Output Depth Register	1	W	1	No	None	No

Related topics

Registers

Relative addressing (HLSL PS reference)

Article • 06/18/2021 • 2 minutes to read

The [] syntax can be used only in register types that can be relatively addressed in certain shader models. The supported forms of [] syntax are listed as follows:

Where:

- "R" denotes any register type that can be relatively addressed.
- "A" denotes any register that can be used as an index to relatively address other registers.
- $n_0 - n_i$, $m_0 - m_j$, and k are integers ≥ 0 .

[] syntax	Effective index	Examples
$R[A + m_0 + \dots + m_j]$	$A + m_0 + \dots + m_j$	$c[a0.x + 3 + 7]$
$R[k] (= Rk)$	k	$c[10] (= c10)$
$R[A]$	A	$c[a0.y]$
$Rk[n_0 + \dots + n_i + A + m_0 + \dots + m_j]$	$A + k + n_0 + \dots + n_i + m_0 + \dots + m_j$	$c8[3 + 2 + a0.w + 5 + 6 + 1]$
$R[n_0 + \dots + n_i + A + m_0 + \dots + m_j]$	$A + n_0 + \dots + n_i + m_0 + \dots + m_j$	$c[2 + 1 + aL + 3 + 4 + 5]$
$Rk[A]$	$A + k$	$c12[aL], c0[a0.z]$
$Rk[A + m_0 + \dots + m_j]$	$A + k + m_0 + \dots + m_j$	$v1[aL + 4 + 8]$
$R[n_0 + \dots + n_i + A]$	$A + n_0 + \dots + n_i$	$o[3 + 1 + aL]$
$Rk[n_0 + \dots + n_i + A]$	$A + k + n_0 + \dots + n_i$	$o1[2 + 1 + 3 + aL]$

The registers are available in the following versions:

Register type	Pixel Shader Versions
loop counter: aL on input registers	ps_3_0 and higher

Related topics

Registers

Input Color Register

Article • 08/23/2019 • 2 minutes to read

Pixel shader input register containing vertex color.

Syntax

```
dcl v#.writeMask
```

where:

- [dcl - \(sm2, sm3 - ps asm\)](#) is a register declaration instruction.
- v is an input register and # is the register number. The number of registers allowed is determined by the shader version.
- writeMask determines which components (up to four) are written. Valid components are: (x,y,z,w) or (r,g,b,a).

Remarks

Color registers are read-only registers. Each register contains four-component RGBA values iterated from input vertices. They have lower precision than most registers, guaranteed to have 8 bits of unsigned data in the range (0, +1). You cannot use more than one in a single instruction.

Related topics

[Registers](#)

[ps_1_1_ps_1_2_ps_1_3_ps_1_4 Registers](#)

[ps_2_0 Registers](#)

[ps_2_x Registers](#)

[ps_3_0 Registers](#)

Constant Boolean register (HLSL PS reference)

Article • 08/19/2020 • 2 minutes to read

This register is a collection of bits used in static flow control instructions (for example, [if](#) [bool - ps - else - ps - endif - ps](#)). There are 16 of them, therefore, a shader can have 16 independent branch conditions. They can be set using [defb - ps](#) or [SetPixelShaderConstantB](#).

The behavior of shader constants has changed between Direct3D 8 and Direct3D 9.

- For Direct3D 9, constants set with defx assign values to the shader constant space. The lifetime of a constant declared with defx is confined to the execution of that shader only. Conversely, constants set using the APIs SetXXXShaderConstantX initialize constants in global space. Constants in global space are not copied to local space (visible to the shader) until SetxxxShaderConstants is called.
- For Direct3D 8, constants set with defx or the APIs both assign values to the shader constant space. Each time the shader is executed, the constants are used by the current shader regardless of the technique used to set them.

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_sw	2_x	3_0	3_sw
Constant Boolean register						x	x	x	

Related topics

[Registers](#)

Constant float register (HLSL PS reference)

Article • 08/19/2020 • 2 minutes to read

Pixel shader input register for a 4D floating-point constant.

They can be set using [def - ps](#) or [SetPixelShaderConstantF](#).

The behavior of shader constants has changed between Direct3D 8 and Direct3D 9.

- For Direct3D 9, constants set with defx assign values to the shader constant space. The lifetime of a constant declared with defx is confined to the execution of that shader only. Conversely, constants set using the APIs SetXXXShaderConstantX initialize constants in global space. Constants in global space are not copied to local space (visible to the shader) until SetxxxShaderConstants is called.
- For Direct3D 8, constants set with defx or the APIs both assign values to the shader constant space. Each time the shader is executed, the constants are used by the current shader regardless of the technique used to set them.

Examples

Here is an example declaring two floating-point constants within a shader.

```
def c40, 0.0f,0.0f,0.0f,0.0f;
```

These constants are loaded every time [SetPixelShader](#) is called.

If you are setting constant values with the API, there is no shader declaration required.

Related topics

[Registers](#)

Constant integer register (HLSL PS reference)

Article • 08/19/2020 • 2 minutes to read

Constant integer registers are used only by [loop - ps](#) and [rep - ps](#).

They can be set using [defi - ps](#) or [SetPixelShaderConstant1](#).

When used as an argument to the [loop - ps](#) instruction:

- `.x` is the iteration count. ([rep - ps](#) uses only this component).
- `.y` is the initial value for the loop counter.
- `.z` is the increment step for the loop counter.

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_sw	2_x	3_0	3_sw
Constant Integer Register						x	x	x	

The behavior of shader constants has changed between Direct3D 8 and Direct3D 9.

- For Direct3D 9, constants set with `defx` assign values to the shader constant space. The lifetime of a constant declared with `defx` is confined to the execution of that shader only. Conversely, constants set using the APIs `SetXXXShaderConstantX` initialize constants in global space. Constants in global space are not copied to local space (visible to the shader) until `SetxxxShaderConstants` is called.
- For Direct3D 8, constants set with `defx` or the APIs both assign values to the shader constant space. Each time the shader is executed, the constants are used by the current shader regardless of the technique used to set them.

Related topics

[Registers](#)

Loop counter register (HLSL PS reference)

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

The only register in this bank is the current loop counter (aL) register. It automatically gets incremented in each execution of the [loop - ps/endloop - ps](#) block. So it can be used in the block for relative addressing if needed and is invalid to use it outside the loop.

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_sw	2_x	3_0	3_sw
Loop Counter Register						x	x		

Related topics

[Registers](#)

Predicate register (HSL PS reference)

Article • 11/23/2019 • 2 minutes to read

This pixel shader output register contains a per-channel boolean value.

A predicate register is supported by the following versions:

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_sw	2_x	3_0	3_sw
predicate register						x	x	x	

Here are the register properties.

Register type	Count	R/W	# Read ports	# Reads/inst	Dimension	RelAddr	Defaults	Requires DCL
Predicate(p)	1	R/W	1	1	4	N/A	None	N

The predicate register can be modified with [setp_comp - ps](#). There are no default values for this register; an application needs to set the register before it is used.

Related topics

[Registers](#)

Sampler (Direct3D 9 asm-ps)

Article • 08/19/2020 • 2 minutes to read

A sampler is a input pseudo-register for a pixel shader, which is used to identify the sampling stage. There are 16 pixel shader sampling stage registers: s0 to s15. Therefore, up to 16 texture surfaces can be read in a single shader pass. The instructions that use a sampler register are texld and texldp.

Sampler must be declared before use with the [dcl_samplerType \(sm2, sm3 - ps asm\)](#) instruction.

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_sw	2_x	3_0	3_sw
Sampler					x	x	x	x	x

Samplers are pseudo registers because you cannot directly read or write to them.

A sampling unit corresponds to the texture sampling stage, encapsulating the sampling-specific state provided by [SetSamplerState](#). Each sampler uniquely identifies a single texture surface, which is set to the corresponding sampler using the [SetTexture](#). However, the same texture surface can be set at multiple samplers.

At draw time, a texture cannot be simultaneously set as a render target and a texture at a stage.

A sampler might appear as the only argument in the texture load instruction: [texldl - ps](#).

In ps_3_0, if a sampler is used, it needs to be declared at the beginning of the shader program using the [dcl_samplerType \(sm2, sm3 - ps asm\)](#) instruction.

Sampling a texture with a higher dimension than is present in the texture coordinates is illegal. Sampling a texture with a lower dimension than is present in the texture coordinates will ignore the extra texture coordinates.

Related topics

[Registers](#)

Temporary register (HLSL PS reference)

Article • 11/23/2019 • 2 minutes to read

Pixel shader input temporary registers are used to hold intermediate results.

Syntax

```
no declaration is required
```

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_sw	2_x	3_0	3_sw
Temporary Register					x	x	x	x	x

- There are 12 pixel-shader temporary registers: r0 to r11.
- These registers are used for storing intermediate results during computations.
- If a temporary register uses components that are not defined in previous code, shader validation will fail.
- These are at least floating-point precision.
- A maximum of three can be used in a single instruction.

Related topics

[Registers](#)

Texture coordinate register (HLSL PS reference)

Article • 11/23/2019 • 2 minutes to read

Pixel shader input register containing texture coordinates.

Pixel shader versions	1_1	1_2	1_3	1_4	2_0	2_sw	2_x	3_0	3_sw
Texture Coordinate Register					x	x	x	x	x

A texture coordinate register contains texture coordinate data. Before a texture coordinate register is used, it must be declared by a pixel shader declaration. For details about how to declare a texture register, see [dcl - \(sm2, sm3 - ps asm\)](#).

In addition, here are some other properties of texture coordinate registers.

- There are eight pixel-shader texture coordinate registers, t0 to t7.
- These are read-only registers.
- They contain four-component RGBA values iterated from input vertices.
- They contain high precision, high dynamic range data values interpolated from the vertex data. Values are generated with perspective-correct interpolation. Data is floating-point precision, and is signed.
- There is a maximum of one in a single instruction.
- Multiple reads of a texture coordinate register within a shader must use identical [Destination Register Write Mask](#).
- The optional partial precision modifier [_pp] applies to dependent reads. This is because partial precision affects arithmetic operations involving the texture coordinate register. It will not affect the precision of texture address instructions because it does not affect the texture coordinate iterators.

Related topics

[Registers](#)

Output Color Register

Article • 08/19/2020 • 2 minutes to read

The pixel shader color output registers (`oC#`) are write-only registers that output results to multiple render targets.

Syntax

```
oC#
```

Where:

Name	Description
<code>oC0</code>	render target #0
<code>oC1</code>	render target #1
<code>oC2</code>	render target #2
<code>oC3</code>	render target #3

Remarks

- If `oCn` is written but there is no corresponding render target, then this write to `oCn` is ignored.
- The render states `D3DRS_COLORWRITEENABLE`, `D3DRS_COLORWRITEENABLE1`, `D3DRS_COLORWRITEENABLE2` and `D3DRS_COLORWRITEENABLE3` determine which components of `oCn` ultimately get written to the render target (after blend, if applicable). If the shader writes some but not all of the components defined by the `D3DRS_COLORWRITEENABLE*` render states for a given `oCn` register, then the unwritten channels produce undefined values in the corresponding render target. If none of the components of an `oCn` are written, the corresponding render target must not be updated at all (as stated above), so the `D3DRS_COLORWRITEENABLE*` render states do not apply.

Shader Model 2 Restrictions

- oCn can only be written with the [mov - ps](#) instruction.
- oC0 must be always written in the shader.
- No source swizzle (except .xyzw = default swizzle) or source modifier is allowed when writing to any oCn.
- No destination write mask (except .xyzw = default mask) or instruction modifier is allowed when writing to any oCn.
- If oCn is written, then oC0 - oCn-1 must also be written. For example, to write to oC2, you must also write to oC0 and oC1.
- At most one write to any oC# per shader is allowed.
- For ps_2_x and ps_3_0, you cannot write to oC# and oD# registers within dynamic flow control or predication (writes to oC# inside static flow control is fine).

Shader Model 3 Restrictions

- For ps_3_0, output registers oC# and oD# can be written any number of times. The output of the pixel shader comes from the contents of the output registers at the end of shader execution. If a write to an output register does not happen, perhaps due to flow control or if the shader just did not write it, the corresponding rendertarget is also not updated. If a subset of the channels in an output register are written, then undefined values will be written to the remaining channels.
- For ps_3_0, the oC# registers can be written with any writemasks.
- For ps_2_x and ps_3_0, you cannot write to oC# and oD# registers within dynamic flow control or predication (writes to oC# inside static flow control is fine).
- You may not perform any gradient calculations (or operations that implicitly invoke gradient calculations such as [texld - ps_2_0 and up](#), [texldb - ps](#), [texldp - ps](#)) inside of flow control statements whose branching conditions vary on a per-primitive basis (ie: dynamic flow control instructions). Instruction predication is not considered dynamic flow control.

Related topics

[Registers](#)

[Multiple Render Targets \(Direct3D 9\)](#)

Output Depth Register

Article • 08/23/2019 • 2 minutes to read

The pixel shader output depth register (`oDepth`) is a write-only scalar register with the range [0..1] that returns a new depth value for a depth test against the depth-stencil buffer.

Syntax

oDepth

Where:

Name	Description
<code>oDepth</code>	New depth value for a depth test against the depth-stencil buffer

It is important to be aware that writing to `oDepth` causes the loss of any hardware-specific depth buffer optimization algorithms (i.e. hierarchical Z) which accelerate depth test performance.

Replicate source swizzle (.x | .y | .z | .w) is required when writing to `oDepth`. Explicit write masks are not allowed.

Writing to the `oDepth` register replaces the interpolated depth value (and ignores any depth bias/slopescale renderstates). If no depth buffer has been created or attached to the device, then write to `oDepth` is ignored.

If you are multisampling and write to `oDepth`, since the pixel shader only runs once per pixel, your depth value is replicated for all covered sub-sample locations. The depth test still happens per-sample, but you don't have a per-sample depth value going into the comparison from the pixel shader like you would have if you didn't write `oDepth`.

If an application has a w-buffer set as its depth buffer, then it needs to take that into account while writing to `oDepth`. It potentially needs to send w-range information to the pixel shader and compute the w-range to scale the w-values written out to `oDepth`.

ps_2_0 and ps_2_x Restrictions

- oDepth can only be written with the [mov - ps](#) instruction and can only be done once.
- No source modifier is allowed when writing to oDepth.
- No instruction modifier is allowed when writing to oDepth.
- No writing to oDepth from within a flow control construct, or when using predication.

ps_3_0 Restrictions

- For ps_3_0, output registers oC# and oD# can be written any number of times. The output of the pixel shader comes from the contents of the output registers at the end of shader execution. If a write to an output register does not happen, perhaps due to flow control or if the shader just did not write it, the corresponding render target is also not updated. If a subset of the channels in an output register are written, then undefined values will be written to the remaining channels.
- You can write to oDepth within flow control or predication as long as all possible paths eventually write into the register.
- You may not perform any gradient calculations (or operations that implicitly invoke gradient calculations such as [texld - ps_2_0 and up](#), [texldb - ps](#), [texldp - ps](#)) inside of flow control statements whose branching conditions vary on a per-primitive basis (ie: dynamic flow control instructions). Instruction predication is not considered dynamic flow control.

Related topics

[Registers](#)

Shader Model 4 Assembly

Article • 03/08/2023 • 2 minutes to read

Shader Model 4 requires you to program shaders in HLSL. However, the shader compiler compiles the HLSL code into assembly that runs on the device. If you are using PIX for Windows to debug your shaders, you can choose to display shader code in either HLSL or assembly. This section lists the Shader Model 4 and Shader Model 4.1 assembly instructions that you may encounter when debugging a shader.

Instruction Modifiers

[add](#)

[and](#)

[break](#)

[breakc](#)

[call](#)

[callc](#)

[case](#)

[continue](#)

[continuec](#)

[cut](#)

[dcl_constantBuffer](#)

[dcl_globalFlags](#)

[dcl_immediateConstantBuffer](#)

[dcl_indexableTemp](#)

[dcl_indexRange](#)

[dcl_input](#)

[dcl_input_sv](#)

[dcl_input vPrim](#)

[dcl_maxOutputVertexCount](#)

[dcl_output](#)

[dcl_output oDepth](#)

[dcl_output_sgv](#)

[dcl_output_siv](#)

[dcl_outputTopology](#)

[dcl_resource](#)

[dcl_sampler](#)

[dcl_temps](#)

[default](#)

[deriv_rtx](#)

[deriv_rty](#)

discard
div
dne
dp2
dp3
dp4
else
emit
emitThenCut
endif
endloop
endswitch
eq
exp
frc
ftoi
ftou
ge
iadd ieq
if
ige
ilt
imad
imin
imul
ine
ineg
ishl
ishr
itof
label
ld
log
loop
lt
mad
max
min
mov
movc

```
mul
ne
nop
not
or
resinfo
ret
retc
round_ne
round_ni
round_pi
round_z
rsq
sample
sample_b
sample_c
sample_c_lz
sample_d
sample_l
sincos
sqrt
switch
udiv
uge
ult
umad
umax
umin
umul
ushr
utof
xor
```

Shader Model 4.1 Assembly

Shader Model 4.1 supports all of the Shader Model 4.0 instructions and the following additional instructions:

```
gather4
ld2dms
lod
```

[sampleinfo](#)

[samplepos](#)

Related topics

[Asm Shader Reference](#)

[Shader Model 4](#)

Instruction modifiers (HLSL reference)

Article • 11/23/2019 • 2 minutes to read

Instruction modifiers affect the result of the instruction before it is written to the register. Shader Model 4 and Shader Model 4.1 support the following instruction modifiers.

Source Operand Modifiers

- [Absolute Value](#)
- [Negate](#)

Instruction Result Modifiers

- [Saturate](#)

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

Absolute Value

Article • 11/20/2019 • 2 minutes to read

Take the absolute value of a source operand used in an arithmetic operation.

_abs

This modifier is used for single and double precision floating point and instructions only. The **_abs** modifier forces the sign of the number(s) on the source operand positive, including on INF values.

Applying **_abs** on NaN preserves NaN, although the particular NaN bit pattern that results is not defined.

When **_abs** is combined with the **negate** modifier, the combination forces the sign to be negative, as if the **_abs** modifier is applied first, then the **negate**.

Minimum Shader Model

This modifier is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Instruction Modifiers](#)

Negate

Article • 11/20/2019 • 2 minutes to read

Flips the sign of the value of a source operand used in an arithmetic operation.

-

For single and double precision floating point and instructions, the **negate** modifier flips the sign of the number(s) in the source operand, including on INF values. Applying **negate** on NaN preserves NaN, although the particular NaN bit pattern that results is not defined.

For integer instructions, the **negate** modifier takes the 2's complement of the number(s) in the source operand.

Minimum Shader Model

This modifier is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Instruction Modifiers](#)

Saturate (HLSL reference)

Article • 11/23/2019 • 2 minutes to read

Clamps the result of a single or double precision floating point arithmetic operation to [0.0f...1.0f] range.

`_sat`

The **saturate** instruction result modifier performs the following operation on the result values(s) from a floating point arithmetic operation that has `_sat` applied to it:

`min(1.0f, max(0.0f, value))`

where `min()` and `max()` in the above expression behave in the way [min](#), [max](#), [dmin](#), or [dmax](#) operate.

`_sat(NaN)` returns 0, by the rules for min and max.

Minimum Shader Model

This modifier is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Instruction Modifiers](#)

add (sm4 - asm)

Article • 04/26/2021 • 2 minutes to read

Component-wise add of 2 vectors.

```
add[_sat] dest[.mask], [-]src0[_abs][.swizzle], [-]src1[_abs][.swizzle]
```

Item	Description
<i>dest</i>	[in] The address of the result of the operation.
<i>src0</i>	[in] The vector to add to <i>src1</i> .
<i>src1</i>	[in] The vector to add to <i>src0</i> .

Remarks

The following table shows the results obtained when executing the instruction with various classes of numbers, assuming that neither overflow or underflow occurs. F means finite real number.

src0 src1->	-inf	-F	-denorm	-0	+0	denorm	+F	+inf	NaN
-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf	NaN	NaN
-F	-inf	-F	src0	src0	src0	src0	+F or +0	+inf	NaN
-denorm	-inf	src1	-0	-0	+0	+0	src1	+inf	NaN
-0	-inf	src1	-0	-0	+0	+0	src1	+inf	NaN
+0	i-inf	src1	+0	+0	+0	+0	src1	+inf	NaN
+denorm	-inf	src1	+0	+0	+0	+0	src1	+inf	NaN
+F	-inf	+F or +0	src0	src0	src0	src0	+F	+inf	NaN
+inf	NaN	+inf	+inf	+inf	+inf	+inf	+inf	+inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

and (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Bitwise AND.

and dest[.mask], src0[.swizzle], src1[.swizzle]

Item	Description
<i>dest</i>	[in] The address of the result of the operation.
<i>src0</i>	[in] The 32-bit value to AND with <i>src1</i> .
<i>src1</i>	[in] The 32-bit value to AND with <i>src0</i> .

Remarks

Component-wise logical **AND** of each pair of 32-bit values from *src0* and *src1*. The 32-bit results are placed in *dest*.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

break (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Moves the point of execution to the instruction after the next [endloop](#) or [endswitch](#).

break

Remarks

The token format contains the offset of the corresponding `endloop` or `endswitch` instruction in the Shader as a convenience.

The following example shows the **break** instruction.

```
loop
    // example of termination condition
    if_nz r0.x
        break
    endif
    ...
endloop
```

This instruction must appear within a `loop/endloop` or in a case in a `switch/endswitch`.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes

Shader Model	Supported
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

breakc (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Conditionally moves the point of execution to the instruction after the next [endloop](#) or [endswitch](#).

breakc{_z|_nz} src0.select_component

Item	Description
<i>src0</i>	[in] The component on which to test the condition.

Remarks

The token format contains the offset of the corresponding `endloop` instruction in the Shader as a convenience.

The following example shows the `breakc` instruction.

```
loop
    // example of termination condition
    breakc_z r0.x // break if all bits in r0.x are 0
    breakc_nz r1.x // break if any bit in r1.x is nonzero
    ...
endloop
```

This instruction must appear within a `loop/endloop` or `switch/endswitch`.

The 32-bit register supplied by *src0* is tested at a bit level. If any bit is nonzero, `breakc_nz` will perform the break. If all bits are zero, `breakc_z` will perform the break.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

call (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Calls a subroutine marked by where the label `l#` appears in the program.

`call l#`

Item	Description
<code>l#</code>	[in] The label of the subroutine.

Remarks

When a `ret` is encountered, return execution to the instruction after this call.

The token format contains the offset of the corresponding label in the Shader as a convenience.

The following example shows the call instruction.

```
...
call 13
...
ret
label 13
...
retc_nz r0.x
...
ret
```

Restrictions

- Subroutines can nest 32 deep.
- The return address stack is managed transparently by the implementation.
- If there are already 32 entries on the return address stack and a `call` is issued, the call is skipped over.
- There is no automatic parameter stack. The application can use an indexable temporary register array (`x#[]`) to manually implement a stack. However, the subroutine call return addresses are not visible and are orthogonal to any manual stack management done by the application.

- Indexing of the $l\#$ parameter is not permitted.
- Recursion is not permitted.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

callc (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Conditionally calls a subroutine marked by where the label l# appears in the program.

```
callc{_z|_nz} src0.select_component, l#
```

Item	Description
src0	[in] The component on which to test the condition.
l#	[in] The label of the subroutine.

Remarks

When a [ret](#) is encountered, return execution to the instruction after this call.

The token format contains the offset of the corresponding label in the Shader as a convenience.

The following example shows the call instruction.

```
...
callc_z r1.y, 13 // if all bits in r0.x are 0, call 13
callc_nz r2.z, 13 // if any bit in r0.x is nonzero, call 13
...
ret
label 13
...
retc_nz r0.x
...
ret
```

Restrictions

- Subroutines can nest 32 deep.
- The return address stack is managed transparently by the implementation.
- If there are already 32 entries on the return address stack and a [call](#) is issued, the call is skipped over.

- There is no automatic parameter stack. The application can use an indexable temporary register array ($x\#[\cdot]$) to manually implement a stack. However, the subroutine call return addresses are not visible and are orthogonal to any manual stack management done by the application.
- Indexing of the $l\#$ parameter is not permitted.
- The 32-bit register supplied by $src0$ is tested at a bit level. If any bit is nonzero, `callc_nz` will perform the call. If all bits are zero, `callc_z` will perform the call.
- Recursion is not permitted.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

case (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

A label in a switch instruction.

case [32-bit immediate]

Remarks

Because falling through **cases** is valid only if there is no code added, multiple **cases** (including **default**) can share the same code block.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

Shader Model 4 Assembly (DirectX HLSL)

continue (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Continues execution at the beginning of the current loop.

continue

Remarks

continue can be used only inside a [loop](#) or [endloop](#).

The following example shows how to use the **continue** instruction.

```
loop
    if_na r0.x
        break
    endif
    if_z r1.x
        ...
        continue
    endif
    ...
endloop
```

The token format contains the offset of the corresponding loop instruction in the Shader as a convenience.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

continuec (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Conditionally continues execution at the beginning of the current loop.

continuec{_z|_nz} src0.select_component

Term	Description
<i>src0</i>	[in] The component against which to test the condition.

Remarks

`continuec` can be used only inside a [loop](#) or [endloop](#).

The following example shows how to use the `continuec` instruction.

```
loop
    if_na r0.x
        break
    endif
    continuec_z r1.x // if all bits of r1.x are zero then
                      // continue at beginning of loop.
    ...
    continuec_nz r3.y // if any bit in r3.y is set then
                      // continue at beginning of loop.

    ...
endloop
```

The token format contains the offset of the corresponding loop instruction in the Shader as a convenience.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

cut (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Geometry Shader instruction that completes the current primitive topology (if any vertices have been emitted), and starts a new topology of the type declared by the Geometry Shader.

cut

Remarks

When **cut** is executed, the first thing that happens is that any previously emitted topology by the Geometry Shader invocation is completed. If not enough vertices were emitted for the previous primitive topology, they are discarded. Because the only available output topologies for the Geometry Shader are pointlist, linestrip, and trianglestrip, there are never any leftover vertices upon **cut**.

After the previous topology, if any, is completed, **cut** causes a new topology to begin, using the topology declared as the Geometry Shader output.

Restrictions

- The **cut** instruction applies only to the Geometry Shader.
- **cut** can appear any number of times in the Geometry Shader, including within flow control.
- If the Geometry Shader ends and vertices have been emitted, the topology they are building is completed, as if a **cut** was executed as the last instruction.
- If streams have been declared, then [cut_stream](#) must be used instead of **cut**.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
	x	

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

dcl_constantBuffer (sm4 - asm)

Article • 08/20/2021 • 2 minutes to read

Declares a shader constant buffer.

dcl_constantBuffer *cbN[size]*, *AccessPattern*

Item	Description
<i>cbN[size]</i>	[in] A shader constant buffer where N is an integer that denotes the constant-buffer-register number and size is an integer that denotes the number of elements in the buffer.
<i>AccessPattern</i>	[in] The way that the buffer will be accessed by shader code, which is one of the following:
Name	Description
immediateIndexed	Index the buffer with a literal value.
dynamic_indexed	Index the buffer with the result of an evaluated expression.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

This instruction is included to aid in debugging a shader in assembly; you cannot author a shader in assembly language using Shader Model 4.

Example

This example declares a constant buffer for register cb0, which has 19 elements. These elements are accessed with a literal index.

```
dcl_constantbuffer cb0[19], immediateIndexed
```

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

dcl_globalFlags (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Declares shader global flags.

dcl_globalFlags *flags*

flags

[in] A global shader flag. There is currently one flag defined.

- REFACTORING_ALLOWED - Permits the driver to reorder arithmetic operations for optimization, as shown here.

```
// Original code  
a = b*c + b*d + b*e + b*f  
  
// Reordered code  
a = b*(c + d + e + f)  
// or  
a = dot4((b,b,b,b), (c,d,e,f))
```

ⓘ Note

Reordering arithmetic operations may generate different results.

Remarks

This optional instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

This instruction is included to aid in debugging a shader in assembly; you cannot author a shader in assembly language using Shader Model 4.

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

dcl_immediateConstantBuffer (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Declares a shader immediate-constant buffer.

```
dcl_immediateConstantBuffer value(s)
```

value(s)

[in] The buffer must contain at least one value, but not more than 4096 values.

Remarks

A shader is allowed one immediate-constant buffer. An immediate-constant buffer is accessed just like a constant buffer with dynamic indexing.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

This instruction is included to aid in debugging a shader in assembly; you cannot author a shader in assembly language using Shader Model 4.

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no

Shader Model	Supported
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

dcl_indexableTemp (sm4 - asm)

Article • 03/08/2023 • 2 minutes to read

Declares an indexable, temporary register.

dcl_indexableTemp xN[size], ComponentCount

Item	Description
xN	[in] A temporary indexable register. <ul style="list-style-type: none">• <i>N</i> is an integer that identifies the register number.• <i>[size]</i> is an optional integer value. The number of elements in the register array.
ComponentCount	[in] An optional integer value. The number of components in the register array.

A register contains enough space for a 32-bit four-component value; the number of elements in the array of temporary registers (indexable and [non-indexable](#)) cannot exceed 4096.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

This instruction is included to aid in debugging a shader in assembly; you cannot author a shader in assembly language using Shader Model 4.

Example

Here are some examples of the code generated for indexable registers.

```
dcl_indexableTemp x0[23], 2 ; // An indexable array of 23, 2-component, 32-bit elements
dcl_indexableTemp x1[16], 4 ; // An indexable array of 16, 4-component, 32-bit elements
```

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

dcl_indexRange (sm4 - asm)

Article • 03/08/2023 • 2 minutes to read

Declares a range of registers that will be accessed by index (an integer computed in the shader).

dcl_indexRange minRegisterM, maxRegisterN

Item	Description
<i>minRegisterM</i>	[in] The first register to access by index. <ul style="list-style-type: none">• <i>minRegister</i> is either v for a vertex or pixel shader input register, or o for a vertex shader output register.• <i>M</i> is an integer that denotes the register number.
<i>maxRegisterN</i>	[in] The last register to access by index. Same form as <i>minRegister</i> except <i>N</i> is the register number.

The following restrictions apply to all registers:

- The min and max registers must be the same type and have the same component masks (if masks are declared).
- A register may have multiple index ranges, as long as they do no not overlap.
- The min register number must be less than the max register number.
- An index register cannot contain a [system-value](#).
- Indexing a register outside of the max index declaration produces undefined results.

Pixel shader input registers must use the same interpolation mode; pixel shader output registers are not indexable.

A geometry shader input register has two dimensions (vertex axis, attribute axis); the index range applies only to the attribute axis because the vertex axis is always fully indexable.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

This instruction is included to aid in debugging a shader in assembly; you cannot author a shader in assembly language using Shader Model 4.

Example

Here is an example.

```
dcl_indexRange v1, v3  
dcl_indexRange v4, v9
```

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

dcl_input (sm4 - asm)

Article • 03/08/2023 • 2 minutes to read

Declares a shader-input register.

dcl_input vN[.mask][, interpolationMode]

Item	Description
vN	[in] A vertex data register. <ul style="list-style-type: none">• N is an integer that identifies the register number.• $[\cdot\text{mask}]$ is an optional component mask (.xyzw) that specifies which of the register components to use.
<i>interpolationMode</i>	[in] Optional. The interpolation mode, which is only honored on pixel shader input registers. It can be one of the following values: <ul style="list-style-type: none">• constant - do not interpolate between register values.• linear - interpolate linearly between register values.• linearCentroid - same as linear but centroid clamped when multisampling.• linearNoperspective - same as linear but with no perspective correction.• linearNoperspectiveCentroid - same as linear, centroid clamped when multisampling, no perspective correction.

Interpolation Notes

By default, vertex attributes are interpolated from a pixel center when performing multisample antialiasing. If a pixel center is not covered, an attribute is extrapolated to a pixel center before interpolation.

For a pixel that is not fully covered or an attribute that does not cover a pixel center, you can specify centroid sampling which forces sampling to occur somewhere within the covered area of the pixel. Because a sample mask (if used) is applied before the centroid is computed, any sample location masked out by the sample mask cannot be chosen as a centroid location.

This instruction applies to the following shader stages:

Vertex Shader

Geometry Shader

Pixel Shader

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

To identify the input as a system value, use [dcl_input_sv \(sm4 - asm\)](#).

This instruction is included to aid in debugging a shader in assembly; you cannot author a shader in assembly language using Shader Model 4.

Example

Here are some examples.

```
dcl_input v3.xyz  
dcl_input v0.x, linearCentroid
```

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

dcl_input_sv (sm4 - asm)

Article • 03/08/2023 • 2 minutes to read

Declares a shader-input register that expects a [system-value](#) to be provided from a preceding stage.

dcl_input_sv vN[.mask], systemValueName[, interpolationMode]

Item	Description
vN	<p>[in] A vertex data register.</p> <ul style="list-style-type: none">• N is an integer that identifies the register number.• $[.mask]$ is an optional component mask (.xyzw) that specifies which of the register components to use.
systemValueName	[in] The system-value name which is a string (see system-value semantics) without the "SV_" prefix.
interpolationMode	[in] Optional. The interpolation mode which affects how values are calculated during rasterization; the mode is only used by a pixel shader. It can be one of the following values: <ul style="list-style-type: none">• constant - do not interpolate between register values.• linear - interpolate linearly between register values.• linearCentroid - same as linear but centroid clamped when multisampling.• linearNoperspective - same as linear but with no perspective correction.• linearNoperspectiveCentroid - same as linear but with no perspective correction and centroid clamped when multisampling.

A component mask for a system-value declaration can be any appropriate subset of [xyzw]; declarations may not overlap (each declaration must follow the sequence xyzw). When declaring scalar system values (clip distance and cull distance for example), you can declare multiple system values in a single register. If you do so, make sure other modifiers like the interpolation modes match.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

This instruction is included to aid in debugging a shader in assembly; you cannot author a shader in assembly language using Shader Model 4.

Example

Here are some examples:

```
// valid
dcl_input v0.y, linear
dcl_input_sv v0.w, clipDistance
dcl_input_sv v0.z, cullDistance
```

```
// invalid declarations
dcl_input v0.y, linear
dcl_input_sv v0.x, clipDistance // the y component was previously declared,
this declaration must use
                                // either the z or w component

dcl_input v0.y, linearNoPerspective           // the interpolation
mode is linear-no-perspective
dcl_input_sv v0.z, renderTargetArrayIndex, constant // the interpolation
modes is constant
                                // the interpolation
modes must match
```

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

dcl_input vPrim (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Declares that a geometry shader uses its scalar input-register vPrim.

dcl_input vPrim

Item	Description
vPrim	[in] A 32-bit scalar, which can be applied to each interior primitive in a geometry shader.

The scalar cannot be applied to any adjacent primitives.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
	x	

This instruction is included to aid in debugging a shader in assembly; you cannot author a shader in assembly language using Shader Model 4.

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

dcl_inputPrimitive (sm4 - asm)

Article • 08/19/2021 • 2 minutes to read

Declares the primitive type for geometry-shader inputs.

dcl_inputPrimitive *type*

Item	Description
<i>type</i>	[in] Input-data primitive type, which is one of the following: <ul style="list-style-type: none">• <i>point</i> - point list• <i>line</i> - line list• <i>triangle</i> - triangle list• <i>line_adj</i> - line list with adjacency data• <i>triangle_adj</i> - triangle list with adjacency data

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
	x	

This instruction is included to aid in debugging a shader in assembly; you cannot author a shader in assembly language using Shader Model 4.

Example

Here is an example.

```
dcl_inputPrimitive triangle
```

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes

Shader Model	Supported
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

dcl_maxOutputVertexCount (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Declares the maximum number of vertices that can be output by a geometry shader.

dcl_maxOutputVertexCount *count*

Item	Description
<i>count</i>	[in] A 32-bit unsigned integer between 1 and n, inclusive.

A geometry shader can output a maximum of 1024 32-bit values. This maximum includes the size of the input data and the size of the data created by the shader.

Here are a few limitations:

- If the number of vertices is reached before the geometry shader finishes executing, the shader terminates.
- A geometry-shader can reach the end of its program before outputting any vertices; this is perfectly legal.
- If you are debugging a geometry shader, you can tell the number of vertices generated by counting the number of emit instructions generated.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
	x	

This instruction is included to aid in debugging a shader in assembly; you cannot author a shader in assembly language using Shader Model 4.

Example

Here are some examples.

Assume input data made up of position (.xyzw) and color (.rgb). Each component consumes one byte; given 7 bytes, the maximum number of vertices the shader can generate would be $1024 / (4 + 3) = 146$.

```
dcl_maxOutputVertexCount 146
```

Assume your geometry shader creates 32 4-component vectors. The maximum number of vertices the shader can generate would be $1024 / (32 * 4) = 8$.

```
dcl_maxOutputVertexCount 8
```

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

dcl_output (sm4 - asm)

Article • 03/08/2023 • 2 minutes to read

Declares a shader-output register.

```
dcl_output oN[.mask]
```

Item	Description
oN	[in] An output data register; <ul style="list-style-type: none">• N is an integer that identifies the register number.• $[.mask]$ is an optional component mask (.xyzw) that specifies which of the register components to use.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

This instruction is included to aid in debugging a shader in assembly; you cannot author a shader in assembly language using Shader Model 4.

Example

Here are some examples.

```
dcl_output o0.xyz  
dcl_output o1  
dcl_output o2.xw
```

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes

Shader Model	Supported
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

dcl_output oDepth (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Declares that a pixel shader will use the output-depth register.

```
dcl_output oDepth
```

The value in the output-depth register is used during a depth comparison (if depth compare is enabled).

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
		x

This instruction is included to aid in debugging a shader in assembly; you cannot author a shader in assembly language using Shader Model 4.

Example

Here are some examples.

```
dcl_output oDepth
```

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes

Shader Model	Supported
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

dcl_output_sgsv (sm4 - asm)

Article • 03/08/2023 • 2 minutes to read

Declares an output register that contains a [system-value](#) parameter.

```
dcl_output_sgsv oN[.mask], systemValueName
```

Item	Description
<code>oN</code>	[in] An output data register; <ul style="list-style-type: none">N is an integer that identifies the register number.<code>[.mask]</code> is an optional component mask (.xyzw) that specifies which of the register components to use.
<code>systemValueName</code>	[in] The system-value name which is a string (see system-value semantics) without the "SV_" prefix.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
		x

This instruction is included to aid in debugging a shader in assembly; you cannot author a shader in assembly language using Shader Model 4.

Example

Here is an example.

```
dcl_output_sgsv o4.x, primitiveID
```

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes

Shader Model	Supported
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

dcl_output_siv (sm4 - asm)

Article • 03/08/2023 • 2 minutes to read

Declares an output register that contains a [system-value](#) parameter.

dcl_output_siv oN[.mask], systemValueName

Item	Description
<code>oN</code>	[in] An output data register; <ul style="list-style-type: none">N is an integer that identifies the register number.$[.mask]$ is an optional component mask (.xyzw) that specifies which of the register components to use.
<code>systemValueName</code>	[in] The system-value name which is a string (see system-value semantics) without the "SV_" prefix.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	

This instruction is included to aid in debugging a shader in assembly; you cannot author a shader in assembly language using Shader Model 4.

Example

Here are some examples.

```
dcl_output o[0].y  
dcl_output_siv o[0].w, clipDistance  
dcl_output_siv o[0].z, cullDistance
```

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
--------------	-----------

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

dcl_outputTopology (sm4 - asm)

Article • 03/08/2023 • 2 minutes to read

Declares the primitive type geometry-shader output data.

dcl_outputTopology *type*

Item	Description
<i>type</i>	[in] An output primitive topology, which is one of the following values: <ul style="list-style-type: none">• <i>pointlist</i>• <i>linestrip</i>• <i>trianglestrip</i>

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
	x	

This instruction is included to aid in debugging a shader in assembly; you cannot author a shader in assembly language using Shader Model 4.

Example

Here is an example.

```
dcl_outputTopology trianglestrip
```

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes

Shader Model	Supported
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

dcl_resource (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Declares a non-multisampled shader-input resource.

```
dcl_resource tN, resourceType, returnType(s)
```

Declares a multisampled shader-input resource.

```
dcl_resource tN, resourceType[size]NN, returnType(s)
```

Item	Description
tN	[in] The texture register, where <i>N</i> is an integer that denotes the register number.
resourceType	[in] Any object type listed in the texture-object page.
resourceType[size]NN	[in] A Texture2D or a Texture2DArray object type (see the texture-object page); <i>size</i> is an optional integer that denotes the number of elements in the array; <i>NN</i> is an integer that denotes the number of multisamples.
returnType(s)	[in] Per-component return type which is one of the following: UNORM, SNORM, SINT, UINT, or FLOAT. The number of return types can be as few as 1 (if all components are the same type), but can be as many as four.

A resource is accessed in HLSL using [load](#); a non-multisampled texture can also be accessed using any of the HLSL [texture object](#) sample methods.

If a resource is bound to a shader stage, the resource format will be validated against the return type.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

This instruction is included to aid in debugging a shader in assembly; you cannot author a shader in assembly language using Shader Model 4.

Example

Here is an example.

```
dcl_resource t3, buffer, UNORM
```

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

dcl_sampler (sm4 - asm)

Article • 08/20/2021 • 2 minutes to read

Declares a sampler register.

dcl_sampler sN, mode

Item	Description
sN	[in] A sampler register, where <i>N</i> is an integer that denotes the register number.
mode	[in] A sampler mode, which constrains which sampler states (listed in the members of D3D10_SAMPLER_DESC) are honored. The modes and states are listed in the following table.

Mode	Sampler States Honored
default	<i>Filter</i> (may not use the <i>_COMPARISON</i> or <i>_TEXT</i> values), <i>AddressU/V/W</i> , <i>MinLOD/MaxLOD</i> , <i>MipLODBias</i> , <i>MaxAnisotropy</i> , <i>BorderColor[4]</i>
comparison	<i>Filter</i> , <i>ComparisonFunction</i> , <i>AddressU/V/W</i> , <i>MinLOD</i> , <i>MaxLOD</i> , <i>MipLODBias</i> , <i>MaxAnisotropy</i> , <i>BorderColor[4]</i>
mono	<i>Filter</i> (must be one of the <i>_TEXT</i> values), <i>MonoFilterWidth</i> , <i>MonoFilterHeight</i> (these two states are global device state), <i>MinLOD</i> , <i>MipLODBias</i> , <i>MaxAnisotropy</i>

The mode restricts the sample instructions that can be used; this table lists the texture-object methods that are supported for each mode.

A Sampler Operating in this Mode	Can Use these Texture-Object Methods
default	Sample , SampleLevel , SampleGrad
comparison	SampleCmp , SampleCmpLevelZero
mono	SampleLevel

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x*

* - Using a sampler in mono mode is supported only in a pixel shader.

This instruction is included to aid in debugging a shader in assembly; you cannot author a shader in assembly language using Shader Model 4.

Example

Here is an example.

```
dcl_sampler s3, default
```

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

dcl_temps (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Declares temporary registers.

dcl_temps N

Item	Description
N	[in] The number of temporary registers.

Each register has space for a 32-bit four-component value. The total number of temporary and [indexable-temporary](#) registers must be less than or equal to 4096.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

This instruction is included to aid in debugging a shader in assembly; you cannot author a shader in assembly language using Shader Model 4.

Example

Here is an example.

```
dcl_temps 10; // Declare r0-r9
```

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes

Shader Model	Supported
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

default (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

An optional label in a [switch](#) statement.

```
default
```

Remarks

This instruction operates just like **default** in C. Falling through is valid only if there is no code added, so multiple cases (including **default**) can share the same code block.

Only one **default** statement is permitted in a **switch** construct.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

deriv_rtx (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Rate of change of contents of each float32 component of *src0* (post-swizzle), with regard to RenderTarget x direction (rtx) or RenderTarget y direction.

```
deriv_rtx[_sat] dest[.mask], [-]src0[_abs][.swizzle],
```

Item	Description
<i>dest</i>	[in] The address of the result of the operation.
<i>src0</i>	[in] The component in the operation.

Remarks

Only a single x,y derivative pair is computed for each 2x2 stamp of pixels.

This operation is hardware dependent.

Reference rasterizer implementation for triangles:

- Pixel Shader always runs Shader over 2x2 quad of pixels in lockstep (even through flow control, masking disabled pixels).
- Quads always have even numbered pixel coordinates (both x and y) for top-left pixel.
- Dummy pixels run off primitive if primitive is too small to fill a 2x2 quad.
- **deriv_rtx** is computed by first choosing 2 pixels: the current pixel and the other pixel with the same y coordinate from the quad. Then, the result is calculated as: *src0*(odd x pixel) - *src0*(even x pixel) [per-component]
- **deriv_rty** is computed by first choosing 2 pixels: the current pixel and the other pixel with the same x coordinate from the quad. Then, the result is calculated as: *src0*(odd y pixel) - *src0*(even y pixel) [per-component]

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
		x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

deriv_rty (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Render-target y equivalent of [deriv_rtx](#).

```
deriv_rty[_sat] dest[.mask], [-]src0[_abs][.swizzle],
```

Item	Description
<i>dest</i>	[in] The address of the result of the operation.
<i>src0</i>	[in] The component in the operation.

Remarks

Only a single x,y derivative pair is computed for each 2x2 stamp of pixels.

This operation is hardware dependent.

Reference rasterizer implementation for triangles:

- Pixel Shader always runs Shader over 2x2 quad of pixels in lockstep (even through flow control, masking disabled pixels).
- Quads always have even numbered pixel coordinates (both x and y) for top-left pixel.
- Dummy pixels run off primitive if primitive is too small to fill a 2x2 quad.
- [deriv_rtx](#) is computed by first choosing 2 pixels: the current pixel and the other pixel with the same y coordinate from the quad. Then, the result is calculated as: $src0(\text{odd } x \text{ pixel}) - src0(\text{even } x \text{ pixel})$ [per-component]
- [deriv_rty](#) is computed by first choosing 2 pixels: the current pixel and the other pixel with the same x coordinate from the quad. Then, the result is calculated as: $src0(\text{odd } y \text{ pixel}) - src0(\text{even } y \text{ pixel})$ [per-component]

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
		x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

discard (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Conditionally flag results of Pixel Shader to be discarded when the end of the program is reached.

```
discard{z|nz} src0.select_component
```

Item	Description
<i>src0</i>	[in] The value that determines whether to discard the current pixel being processed.

Remarks

This instruction flags the current pixel as terminated, while continuing execution, so that other pixels executing in parallel may obtain derivatives if necessary. Even though execution continues, all Pixel Shader output writes before or after the **discard** instruction are discarded.

For **discard_z**, if all bits in *src0.select_component* are zero, then the pixel is discarded.

For **discard_nz**, if any bits in *src0.select_component* are nonzero, then the pixel is discarded.

In addition, the **discard** instruction can be present inside any flow control construct.

Multiple **discard** instructions may be present in a Shader, and if any is executed, the pixel is terminated.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
		x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes

Shader Model	Supported
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

div (sm4 - asm)

Article • 03/08/2023 • 2 minutes to read

Component-wise divide.

div[_sat] dest[.mask], [-]src0[_abs][.swizzle], [-]src1[_abs][.swizzle]

Item	Description
<i>dest</i>	[in] The result of the operation.
<i>src0</i>	[in] The dividend.
<i>src1</i>	[in] The divisor.

Remarks

The following table shows the results obtained when executing the instruction with various classes of numbers, assuming that neither overflow or underflow occurs.

You should note the two allowed implementations of divide: a/b and a*(1/b).

One outcome of this is that there are exceptions to the table below for large denominator values (greater than 8.5070592e+37), where 1/denominator is a denorm. Because implementations may perform divide as a*(1/b), instead of a/b directly, and 1/[large value] is a denorm that could get flushed, some cases in the table would produce different results. For example, (+/-)INF / (+/-)[value > 8.5070592e+37] may produce NaN on some implementations, but (+/-)INF on other implementations

In this table F means finite-real number.

src0	src1	-	-inf	-F	-	-0	+0	+denorm	+F	+inf	Nan
> denorm											
-inf	-inf	-	-inf	-inf	-inf	-inf	-inf	-inf	NaN	NaN	
-F	-inf	-	-F	src0	src0	src0	src0	+F or +0	+inf	NaN	
-denorm	-inf	src1	-	-0	-0	+0	+0	src1	+inf	NaN	
-0	-inf	src1	-	-0	-0	+0	+0	src1	+inf	NaN	
+0	-inf	src1	-	+0	+0	+0	+0	src1	+inf	NaN	
+denorm	-inf	src1	-	+0	+0	+0	+0	src1	+inf	NaN	

src0	src1	-	-inf	-F	-	-0	+0	+denorm	+F	+inf	Nan
> denorm											
+F		-inf	+ -F or + -0	src0	src0	src0	src0	+F	+inf	NaN	
+inf		NaN	+inf	+inf	+inf	+inf	+inf	+inf	+inf	NaN	
NaN		NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

dp2 (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

2-dimensional vector dot-product of components rg, POS-swizzle.

```
dp2[_sat] dest[.mask], [-]src0[_abs][.swizzle], [-]src1[_abs][.swizzle]
```

Item	Description
<i>dest</i>	[in] The address of the result of the operation. $dest = src0.r * src1.r + src0.g * src1.g$
<i>src0</i>	[in] The components in the operation.
<i>src1</i>	[in] The components in the operation.

Remarks

Scalar result replicated to components in write mask.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

dp3 (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

3-dimensional vector dot-product of components rgb, POS-swizzle.

```
dp3[_sat] dest[.mask], [-]src0[_abs][.swizzle], [-]src1[_abs][.swizzle],
```

Item	Description
<i>dest</i>	[in] The result of the operation. $dest = src0.r * src1.r + src0.g * src1.g + src0.b * src1.b$
<i>src0</i>	[in] The components in the operation.
<i>src1</i>	[in] The components in the operation.

Remarks

Scalar result replicated to components in write mask.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

dp4 (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

4-dimensional vector dot-product of components rgba, POS-swizzle.

```
dp4[_sat] dest[.mask], [-]src0[_abs][.swizzle], [-]src1[_abs][.swizzle],
```

Item	Description
<i>dest</i>	[in] The result of the operation. $dest = src0.r * src1.r + src0.g * src1.g + src0.b * src1.b + src0.a * src1.a$
<i>src0</i>	[in] The components in the operation.
<i>src1</i>	[in] The components in the operation.

Remarks

Scalar result replicated to components in write mask.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

else (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Starts an **else** block.

```
else
```

Remarks

The token format contains the offset of the corresponding [endif](#) instruction in the Shader as a convenience.

The following example shows how to use the **else** instruction.

syntax

```
if      // any of the various forms of if* statements
...
else   // (optional)
...
endif
```

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes

Shader Model	Supported
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

emit (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Emit a vertex.

```
emit
```

Remarks

emit causes all declared o# registers to be read out of the Geometry Shader to generate a vertex.

As multiple **emit** calls are issued, primitives are generated.

emit can appear any number of times in a Geometry Shader, including within flow control.

If streams have been declared, you must use [emit_stream](#).

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
	x	

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no

Shader Model	Supported
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

emitThenCut (sm4 - asm)

Article • 03/09/2021 • 2 minutes to read

Equivalent to an [emit](#) command followed by a [cut](#) command.

emitThenCut

Remarks

This command is useful when knowingly outputting the last vertex in a topology.

If streams have been declared, you must use [emitthenCut_stream](#).

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
	x	

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

endif (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Ends an [if](#) statement.

```
endif
```

Remarks

The following example shows how to use the endif instruction.

syntax

```
if      // any of the various forms of if* statements
...
else   // (optional)
...
endif
```

The token format contains the offset of the corresponding [if](#) instruction in the Shader as a convenience.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes

Shader Model	Supported
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

endloop (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Ends a loop statement.

```
endloop
```

Remarks

The following example shows how to use this instruction.

syntax

```
loop
    // example of termination condition
    if_nz r0.x
        break
    endif
    ...
endloop
```

The token format contains the offset of the corresponding loop instruction in the Shader as a convenience.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes

Shader Model	Supported
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

endswitch (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Ends a [switch](#) statement.

```
endswitch
```

Remarks

The token format contains the offset of the corresponding switch instruction in the Shader as a convenience.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

Shader Model 4 Assembly (DirectX HLSL)

eq (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Component-wise vector floating point equality comparison.

```
eq dest[.mask], [-]src0[_abs][.swizzle], [-]src1[_abs][.swizzle]
```

Item	Description
<i>dest</i>	[in] The address of the result of the operation.
<i>src0</i>	[in] The component to compare to <i>src1</i> .
<i>src1</i>	[in] The component to compare to <i>src0</i> .

Remarks

Performs the float comparison ($src0 == src1$) for each component, and writes the result to *dest*.

If the comparison is true, 0xFFFFFFFF is returned for that component. Otherwise 0x00000000 is returned.

Denorms are flushed before comparison (original source registers untouched). +0 equals -0. Comparison with NaN returns false.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes

Shader Model	Supported
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

frc (sm4 - asm)

Article • 04/26/2021 • 2 minutes to read

Component-wise, extract fractional component.

```
frc[_sat] dest[.mask], [-] src0[_abs][.swizzle]
```

Item	Description
<i>dest</i>	[in] The address of the result of the operation. $dest = src0 - \text{round_ni}(src0)$
<i>src0</i>	[in] The component in the operation.

Remarks

The following table shows the results obtained when executing the instruction with various classes of numbers.

src	-inf	-F	-denorm	-0	+0	+denorm	+F	+inf	NaN
dest	NaN	[+0 to 1)	+0	+0	+0	+0	[+0 to 1)	NaN	NaN

F means finite-real number.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes

Shader Model	Supported
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

ftoi (sm4 - asm)

Article • 09/17/2020 • 2 minutes to read

Floating point to signed integer conversion.

ftoi dest[.mask], [-]src0[_abs][.swizzle]

Item	Description
<i>dest</i>	[in] The address of the result of the operation. <i>dest</i> = <code>round_z(src0)</code>
<i>src0</i>	[in] The component to convert.

Remarks

The conversion is performed per component. Rounding is always performed towards zero, following the C convention for casts from float to int. Applications that require different rounding semantics can invoke the **round** instructions before casting to integer.

Inputs are clamped to the range [-2147483648.999f ... 2147483647.999f] prior to conversion, and input NaN values produce a zero result.

Optional negate and absolute value modifiers are applied to the source values before conversion.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes

Shader Model	Supported
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

ftou (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Floating point to unsigned integer conversion.

ftou dest[.mask], [-]src0[_abs][.swizzle]
--

ftoi dest[.mask], [-]src0[_abs][.swizzle]
--

Item	Description
<i>dest</i>	[in] The address of the result of the operation.
<i>src0</i>	[in] The value to convert.

Remarks

The conversion is performed per-component. Rounding is always performed towards zero, following the C convention for casts from float to int.

Applications that require different rounding semantics can invoke the **round** instructions before casting to integer.

Inputs are clamped to the range [0.0f ... 4294967295.999f] prior to conversion, and input NaN values produce a zero result.

Optional negate and absolute value modifiers are applied to the source values before conversion.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes

Shader Model	Supported
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

gather4 (sm4.1 - asm)

Article • 03/08/2023 • 2 minutes to read

Gathers the four texels that would be used in a bi-linear filtering operation and packs them into a single register.

```
gather4[_aoffimmi(u,v)] dest[.mask], srcAddress[.swizzle], srcResource[.swizzle],  
srcSampler.r
```

Item	Description
<i>dest</i>	[in] The address of the result of the operation.
<i>srcAddress</i>	[in] Contains the texture coordinates.
<i>srcResource</i>	[in] A resource register. The swizzle allows the returned values to be swizzled arbitrarily before they are written to <i>dest</i> .
<i>srcSampler</i>	[in] A sampler register. This parameter must have a .r (red) swizzle, which indicates that the value of the R channel is copied to <i>dest</i> .

Remarks

This operation only works with single channel 2D or CubeMap textures. For 2D textures only the addressing modes of the sampler are used and the top level of any mip pyramid is used.

This instruction behaves like the [sample](#) instruction, but a filtered sample is not generated. The four samples that would contribute to filtering are placed into xyzw in counter clockwise order starting with the sample to the lower left of the queried location. This is the same as point sampling with (u,v) texture coordinate deltas at the following locations: (-,+),(+,-),(+,-),(-,-), where the magnitude of the deltas are always half a texel.

For CubeMap textures when a bi-linear footprint spans an edge texels from the neighboring face are used. Corners use the same rules as the [sample](#) instruction; that is the unknown corner is considered the average of the three impinging face corners.

The texture format restrictions that apply to the [sample](#) instructions also apply to the [gather4](#) instruction.

For hardware implementations, optimizations in traditional bilinear filtering that detect samples directly on texels and skip reading of texels that would have weight 0 cannot be leveraged with `gather4`. `gather4` always returns all requested texels.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

ge (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Component-wise vector floating point greater-than-or-equal comparison.

```
ge dest[.mask], [-]src0[_abs][.swizzle], [-]src1[_abs][.swizzle]
```

Item	Description
<i>dest</i>	[in] The address of the result of the operation.
<i>src0</i>	[in] The component to compare to <i>src1</i> .
<i>src1</i>	[in] The component to compare to <i>src0</i> .

Remarks

Performs the float comparison ($src0 \geq src1$) for each component, and writes the result to *dest*.

If the comparison is true, then 0xFFFFFFFF is returned for that component. Otherwise 0x0000000 is returned.

Denorms are flushed before comparison, and original source registers are untouched. +0 equals -0. Comparison with NaN returns false.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes

Shader Model	Supported
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

iadd (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Integer addition.

iadd dest[.mask], [-]src0[.swizzle], [-]src1[.swizzle]

Item	Description
<i>dest</i>	[in] The address of the result of the operation.
<i>src0</i>	[in] The number to be added to <i>src1</i> .
<i>src1</i>	[in] The number to be added to <i>src0</i> .

Remarks

Component-wise add of 32-bit operands *src0* and *src1*, placing the correct 32-bit result in *dest*. No carry or borrow beyond the 32-bit values of each component is performed, so this instruction is not sensitive to the signed-ness of its operands.

Optional negate modifier on source operands takes 2's complement before performing operation.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no

Shader Model	Supported
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

ieq (sm4 - asm)

Article • 03/08/2023 • 2 minutes to read

Component-wise vector integer equality comparison.

ieq dest[.mask], src0[.swizzle], src1[.swizzle]

Item	Description
<i>dest</i>	[in] The address of the result of the operation.
<i>src0</i>	[in] The value to compare with <i>src1</i> .
<i>src1</i>	[in] The value to compare with <i>src0</i> .

Remarks

This instruction performs the integer comparison ($src0 == src1$) for each component, and writes the result to *dest*.

If the comparison is true, then 0xFFFFFFFF is returned for that component. Otherwise 0x00000000 is returned

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no

Shader Model	Supported
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

if (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Branch based on logical OR result.

if{_z|_nz} src0.select_component

Item	Description
<i>src0</i>	[in] Contains the component on which to test the condition.

Remarks

The token format contains the offset of the corresponding endif instruction in the Shader as a convenience.

The following example shows how to use this instruction.

syntax

```
if_z r0.x // if all bits in r0.x are zero
...
else // (optional)
...
endif
if_nz r1.x // if any bit in r0.x is nonzero
...
else // (optional)
...
endif
```

Restrictions

- The source operands (if 4 component vectors) must use a single component selector.
- The 32-bit register supplied by *src0* is tested at a bit level. If any bit is nonzero, **if_z** will be true. If all bits are zero, **if_nz** will be true.
- Flow control blocks can nest up to 64 deep per subroutine (and main). The HLSL compiler will not generate subroutines that exceed this limit. Behavior of control flow instructions beyond 64 levels deep (per subroutine) is undefined.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

ige (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Component-wise vector integer greater-than-or-equal comparison.

ige dest[.mask], src0[.swizzle], src1[.swizzle]

Item	Description
<i>dest</i>	[in] The result of the operation.
<i>src0</i>	[in] The component to compare to <i>src1</i> .
<i>src1</i>	[in] The component to compare to <i>src0</i> .

Remarks

Performs the integer comparison ($src0 \geq src1$) for each component, and writes the result to *dest*.

If the comparison is true, then 0xFFFFFFFF is returned for that component. Otherwise 0x0000000 is returned.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no

Shader Model	Supported
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

ilt (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Component-wise vector integer less-than comparison.

ilt dest[.mask], src0[.swizzle], src1[.swizzle]

Item	Description
<i>dest</i>	The result of the operation.
<i>src0</i>	[in] The value to compare to <i>src1</i> .
<i>src1</i>	[in] The value to compare to <i>src0</i> .

Remarks

Performs the integer comparison ($src0 < src1$) for each component, and writes the result to *dest*.

If the comparison is true, then 0xFFFFFFFF is returned for that component. Otherwise 0x0000000 is returned.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

imad (sm4 - asm)

Article • 03/08/2023 • 2 minutes to read

Signed integer multiply and add.

imad dest[.mask], [-]src0[.swizzle], [-]src1[.swizzle], [-]src2[.swizzle]

Item	Description
<i>dest</i>	[in] The result of the operation.
<i>src0</i>	[in] Value to multiply with <i>src1</i> .
<i>src1</i>	[in] Value to multiply with <i>src0</i> .
<i>src2</i>	[in] Value to add to the product of <i>src0</i> and <i>src1</i> .

Remarks

Component-wise [imul](#) of 32-bit operands *src0* and *src1* (signed), keeping low 32-bits (per component) of the result, followed by an [iadd](#) of *src2*, producing the correct low 32-bit (per component) result. The 32-bit results are placed in *dest*.

Optional negate modifier on source operands takes 2's complement before performing arithmetic operation.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes

Shader Model	Supported
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

imax (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Component-wise integer maximum.

```
imax dest[.mask], [-]src0[.swizzle], [-]src1[.swizzle],
```

Item	Description
<i>dest</i>	[in] The result of the operation. $dest = src0 > src1 ? src0 : src1$
<i>src0</i>	[in] The value to compare to <i>src1</i> .
<i>src1</i>	[in] The value to compare to <i>src0</i> .

Remarks

Optional negate modifier on source operands takes 2's complement before performing operation.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

imin (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Component-wise integer minimum.

```
imin dest[.mask], [ -]src0[.swizzle], [-]src1[.swizzle],
```

Item	Description
<i>dest</i>	[in] The result of the operation. $dest = src0 < src1 ? src0 : src1$
<i>src0</i>	[in] The value to compare to <i>src1</i> .
<i>src1</i>	[in] The value to compare to <i>src0</i> .

Remarks

Optional negate modifier on source operands takes 2's complement before performing operation.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

imul (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Signed integer multiply.

imul destHI[.mask], destLO[.mask], [-]src0[.swizzle], [-]src1[.swizzle]

Item	Description
<i>destHI</i>	[in] The address of the high 32 bits of the result.
<i>destLO</i>	[in] The address of the low 32 bits of the result.
<i>src0</i>	[in] The value to multiply with <i>src1</i> .
<i>src1</i>	[in] The value to multiply with <i>src0</i> .

Remarks

Component-wise multiply of 32-bit operands *src0* and *src1* (both are signed), producing the correct full 64-bit (per component) result. The low 32 bits (per component) are placed in *destLO*. The high 32 bits (per component) are placed in *destHI*.

Either *destHI* or *destLO* may be specified as NULL instead of specifying a register, if the high or low 32 bits of the 64-bit result are not needed.

Optional negate modifier on source operands takes 2's complement before performing arithmetic operation.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes

Shader Model	Supported
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

ine (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Component-wise vector integer not-equal comparison.

ine dest[.mask], src0[.swizzle], src1[.swizzle]

Item	Description
<i>dest</i>	[in] The address of the result of the operation.
<i>src0</i>	[in] Contains the value to compare to <i>src1</i> .
<i>src1</i>	[in] Contains the value to compare to <i>src0</i> .

Remarks

Performs the integer comparison ($src0 \neq src1$) for each component, and writes the result to *dest*.

If the comparison is true, then 0xFFFFFFFF is returned for that component. Otherwise 0x00000000 is returned

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no

Shader Model	Supported
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

ineg (sm4 - asm)

Article • 03/08/2023 • 2 minutes to read

2's complement.

ineg dest[.mask], src0[.swizzle]

Item	Description
<i>dest</i>	[in] The address of the result of the operation.
<i>src0</i>	[in] Contains the values for the operation.

Remarks

This instruction performs component-wise 2's complement of each 32-bit value in *src0*. The 32-bit results are stored in *dest*.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

Shader Model 4 Assembly (DirectX HLSL)

ishl (sm4 - asm)

Article • 03/08/2023 • 2 minutes to read

Shift left.

ishl dest[.mask], src0[.swizzle], src1.select_component

Item	Description
<i>dest</i>	[in] The address of the result of the operation.
<i>src0</i>	[in] Contains the values to be shifted.
<i>src1</i>	[in] Contains the shift amount.

Remarks

This instruction performs a component-wise shift of each 32-bit value in *src0* left by an unsigned integer bit count provided by the LSB 5 bits (0-31 range) in *src1.select_component*, inserting 0. The 32-bit per component results are placed in *dest*. The count is a scalar value applied to all components.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no

Shader Model	Supported
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

ishr (sm4 - asm)

Article • 03/09/2021 • 2 minutes to read

Arithmetic shift right (sign extending).

ishr dest[.mask], src0[.swizzle], src1.select_component

Item	Description
<i>dest</i>	[in] Contains the result of the operation.
<i>src0</i>	[in] Contains the value to be shifted.
<i>src1</i>	[in] Contains the shift amount.

Remarks

This instruction performs a component-wise arithmetic shift of each 32-bit value in *src0* right by an unsigned integer bit count provided by the LSB 5 bits (0-31 range) in *src1.select_component*, replicating the value of bit 31. The 32-bit per component result is placed in *dest*. The count is a scalar value applied to all components.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no

Shader Model	Supported
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

itof (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Signed integer to floating point conversion.

itof dest[.mask], [-]src0[.swizzle]

Item	Description
<i>dest</i>	[in] Contains the result of the operation.
<i>src0</i>	[in] Contains the value to convert.

Remarks

This signed integer-to-float conversion instruction assumes that *src0* contains a signed 32-bit integer 4-tuple. After the instruction executes, *dest* will contain a floating-point 4-tuple.

The conversion is performed per-component.

When an integer input value is too large in magnitude to be represented exactly in the floating point format, rounding to nearest even mode is strongly recommended but not required.

The optional negate modifier on source operand takes 2's complement before performing arithmetic operation.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes

Shader Model	Supported
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

label (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Indicates the beginning of a subroutine.

label l#

Item	Description
<i>l#</i>	[in] The label number.

Remarks

A **label** can only appear directly after a **ret** instruction which is not nested in any flow control statements.

The code before the first **label** in a program is the main program. All subroutines appear at the end of the program, indicated by **label** statements.

The following example shows how to use this instruction.

syntax

```
...
call 13
...
ret
label 13
...
if_nz r0.x
    ret
endif
...
ret
```

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

Id (sm4 - asm)

Article • 11/20/2019 • 3 minutes to read

Fetches data from the specified buffer or texture without any filtering (e.g. point sampling) using the provided integer address. The source data may come from any resource type, other than TextureCube.

Id[_aoffimmi(u,v,w)] dest[.mask], srcAddress[.swizzle], srcResource[.swizzle]

Item	Description
<i>dest</i>	[in] The address of the result of the operation.
<i>srcAddress</i>	[in] The texture coordinates needed to perform the sample.
<i>srcResource</i>	[in] A texture register (t#) which must have been declared identifying which Texture or Buffer to fetch from.

Remarks

This instruction is a simplified alternative to the [sample](#) instruction. Unlike [sample](#), **Id** is also capable of fetching data from buffers. **Id** can also fetch from multi-sample resources (on pixel shader only).

srcAddress provides the set of texture coordinates needed to perform the sample in the form of unsigned integers. If *srcAddress* is out of the range[0...(#texels in dimension -1)], then out-of-bounds behavior is invoked, where **Id** returns 0 in all non-missing components of the format of the *srcResource*, and the default for missing components. An application wishing any more flexible control over out-of-range address behavior should use the [sample](#) instruction instead, as it honors address wrap/mirror/clamp/border behavior defined as sampler state.

srcAddress.a (POS-swizzle) always provides an unsigned integer mipmap level. If the value is out of the range [0...(num miplevels in resource-1)], then out-of-bounds behavior is invoked. If the resource is a buffer, which can not have any mipmaps, then *srcAddress.a* is ignored

srcAddress.gb (POS-swizzle) is ignored for buffers and texture1D (non-Array).

srcAddress.b (POS-swizzle) is ignored for texture1D arrays and texture2Ds.

For texture1D arrays, *srcAddress.g* (POS-swizzle) provides the array index as an unsigned integer. If the value is out of the range of available array indices [0...(array size-1)], then

out-of-bounds behavior is invoked.

For texture2D arrays, *srcAddress.b* (POS-swizzle) provides the array index, otherwise with same semantics as for texture1D.

Fetching from *t#* that has nothing bound to it returns 0 for all components.

Address Offset

The optional [_aoffimmi(u,v,w)] suffix (address offset by immediate integer) indicates that the texture coordinates for the **Id** are to be offset by a set of provided immediate texel space integer constant values. The literal values are a set of 4 bit 2's complement numbers, having integer range [-8,7]. This modifier is defined only for texture1D/2D/3D, including arrays, and not for buffers.

The offsets are added to the texture coordinates, in texel space, relative to the miplevel being accessed by the **Id**.

Address offsets are not applied along the array axis of texture1D/2D arrays.

The _aoffimmi *v,w* components are ignored for texture1Ds.

The _aoffimmi *w* component is ignored for texture2Ds.

Because the texture coordinates for **Id** are unsigned integers, if the offset causes the address to go below zero, it will wrap to a large address, and result in an out of bounds access.

Return Type Control

The data format returned by **Id** to the destination register is determined in the same way as described for the **sample** instruction; it is based on the format bound to the *srcResource* parameter (*t#*).

As with the **sample** instruction, returned values for **Id** are 4-vectors with format-specific defaults for components not present in the format. The swizzle on *srcResource* determines how to swizzle the 4-component result coming back from the texture load, after which .mask on *dest* determines which components in *dest* get updated.

When a 32-bit float value is read by *ld* into a 32-bit register, the bits are untouched; that is, denormal values remain denormal. This is unlike the **sample** instruction.

Miscellaneous Details

As there is no filtering associated with the `Id` instruction, concepts like LOD bias do not apply to `Id`. Accordingly there is no sampler `s#` parameter.

Restrictions

- `srcResource` must be a `t#` register, and not a `TextureCube`. `srcResource` cannot be a `constantbuffer` either, which cannot be bound to `t#` registers.
- Relative addressing on `srcResource` is not permitted.
- `srcAddress` must be a `temp (r#/x#)`, `constant (cb#)` or `input (v#)` register.
- `dest` must be a `temp (r#/x#)` or `output (o*x#)` register.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

Id2dms (sm4.1 - asm)

Article • 03/08/2023 • 3 minutes to read

Reads individual samples out of 2-dimensional multi-sample textures.

```
Id2dms[_aoffimmi(u,v)] dest[.mask], srcAddress[.swizzle], srcResource[.swizzle],  
sampleIndex
```

Item	Description
<i>dest</i>	[in] The address of result of the operation.
<i>srcAddress</i>	[in] The texture coordinates needed to perform the sample.
<i>srcResource</i>	[in] A texture register (t#) which must have been declared identifying which Texture or Buffer to fetch from
<i>sampleIndex</i>	[in] Identifies the samples to read from <i>srcResource</i> (scalar operand).

Remarks

This instruction is a simplified alternative to the [sample](#) instruction. It fetches data from the specified Texture without any filtering (e.g. point sampling) using the provided integer *srcAddress* and *sampleIndex*.

srcAddress provides the set of texture coordinates needed to perform the sample in the form of unsigned integers. If *srcAddress* is out of the range[0...(#texels in dimension -1)], **Id2dms** always returns 0 in all components present in the format of the resource, and defaults (0,0,0,1.0f/0x00000001) for missing components.

sampleIndex does not have to be a literal. The multi-sample count does not have to be specified on the texture resource, and it works with depth or stencil views.

An application wishing any more flexible control over out-of-range address behavior should use the [sample](#) instruction instead, as it honors address wrap/mirror/clamp/border behavior defined as sampler state.

srcAddress.b (post-swizzle) is ignored for Texture2Ds. If the value is out of the range of available array indices [0...(array size-1)], then **Id2dms** always returns 0 in all components present in the format of the resource, and defaults (0,0,0,1.0f/0x00000001) for missing components.

For Texture2D Arrays, *srcAddress.b* (post-swizzle) provides the array index. Otherwise it has the same behavior as Texture2D.

srcAddress.a (post-swizzle) is always ignored. The HLSL compiler will never output anything there.

srcResource is a texture register (*t#*) which must have been declared(22.3.11), identifying which Texture to fetch from.

Fetching from *t#* that has nothing bound to it returns 0 for all components.

Address Offset

The optional *[_aoffimmi(u,v,w)]* suffix (address offset by immediate integer) indicates that the texture coordinates for the **Id2dms** are to be offset by a set of provided immediate texel space integer constant values. The literal values are a set of 4 bit 2's complement numbers, having integer range [-8,7].

The offsets are added to the texture coordinates, in texel space.

Address offsets are not applied along the array axis of Texture1D/2D Arrays.

The *_aoffimmi v,w* components are ignored for Texture1Ds.

The *_aoffimmi w* component is ignored for Texture2Ds.

Because the texture coordinates for **Id2dms** are unsigned integers, if the offset causes the address to go below zero, it will wrap to a large address, and result in an out of bounds access, which like **Id** returns 0 in all components present in the format of the resource, and the defaults (0,0,0,1.0f/0x00000001) for missing components.

Sample Number

Id2dms is available for use on any resource. **Id2dms** operates identically to **Id** except on 2D multisample resources, by using the additional (0-based) *sampleIndex* operand to identify which sample to read from the resource.

The result of specifying a *sampleIndex* that exceeds the number of samples in the resource is undefined, but cannot return data outside of the address space of the device context.

Return Type Control

The data format returned by `Id2dms` to the destination register is determined in the same way as described for the `sample` instruction. It is based on the format bound to the `srcResource` parameter (`t#`).

As with the `sample` instruction, returned values for `Id2dms` are 4-vectors with format-specific defaults for components not present in the format. The swizzle on `srcResource` determines how to swizzle the 4-component result coming back from the texture load, after which `.mask` on `dest` determines which components in `dest` get updated.

When a 32-bit float value is read by `Id2dms` into a 32-bit register, the bits are untouched; that is, denormal values remain denormal. This is unlike the `sample` instruction.

Miscellaneous Details

As there is no filtering associated with this instruction, concepts like LOD bias do not apply. Accordingly there is no `sampler s#` parameter.

Restrictions

- `srcResource` must be a `t#` register, and not a `TextureCube`, `Texture1D` or `Texture1DArray`. `srcResource` cannot be a `ConstantBuffer`, which cannot be bound to `t#` registers.
- Relative addressing on `srcResource` is not permitted.
- `srcAddress` and `sampleIndex` must be a temp (`r#/x#`), constant (`cb#`) or input (`v#`) register.
- `dest` must be a temp (`r#/x#`) or output (`o*#`) register.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes

Shader Model	Supported
Shader Model 4.1	yes
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

lod (sm4.1 - asm)

Article • 11/20/2019 • 2 minutes to read

Returns the level of detail (LOD) that would be used for texture filtering.

lod dest[.mask], srcAddress[.swizzle], srcResource[.swizzle], srcSampler

Item	Description
<i>dest</i>	[in] The address of the results.
<i>srcAddress</i>	[in] A set of texture coordinates.
<i>srcResource</i>	[in] A texture register.
<i>srcSampler</i>	[in] A sampler register.

Remarks

This behaves like the [sample](#) instruction, but a filtered sample is not generated. The instruction computes the following vector (ClampedLOD, NonClampedLOD, 0, 0). NonClampedLOD is a computed LOD value that ignores any clamping from either the sampler or the texture (ie: it can return negative values.) ClampedLOD is a computed LOD value that would be used by the actual [sample](#) instruction. The swizzle on *srcResource* allows the returned values to be swizzled arbitrarily before they are written to the destination.

If there is no resource bound to the specified slot, 0 is returned.

If the sampler is using anisotropic filtering the LOD should correspond to the fractional mip level based on the smaller axis of the elliptical footprint.

This is valid for the following texture types: Texture1D, Texture2D, Texture3D and TextureCube.

The **lod** instruction is not defined when used with a sampler that specifies point mip filtering, specifically, any D3D10_FILTER enum that ends in MIP_POINT. (An example of this would be D3D10_FILTER_MIN_MAG_MIP_POINT.)

This instruction applies to the following shader stages:

Vertex Shader

Geometry Shader

Pixel Shader

Vertex Shader	Geometry Shader	Pixel Shader
		x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

log (sm4 - asm)

Article • 03/08/2023 • 2 minutes to read

Component-wise log base 2.

log[_sat] dest[.mask], [-]src0[_abs][.swizzle]

Item	Description
<i>dest</i>	[in] The address of the result of the operation. $dest = \log_2(src0)$
<i>src0</i>	[in] The value for the operation.

Remarks

Restrictions

- Follows limit theory.
- Error tolerance: If *src0* is [0.5..2], absolute error must be no more than 2^{-21} . If *src0* is (0..0.5) or (2..+INF], relative error must be no more than 2^{-21} .

The following table shows the results obtained when executing the instruction with various classes of numbers, assuming that neither overflow or underflow occurs. F means finite-real number.

src	-inf	-F	-denorm	-0	+0	+denorm	+F	+inf	NaN
dest	NaN	NaN	-inf	-inf	-inf	-inf	F	+inf	NaN

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

loop (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Specifies a loop which iterates until a break instruction is encountered.

```
loop
```

Remarks

loop can iterate indefinitely, although overall execution of the Shader may be forced to terminate after some number of instructions are executed.

Flow control blocks can nest up to 64 deep per subroutine and main. The HLSL compiler will not generate subroutines that exceed this limit. Behavior of control flow instructions beyond 64 levels deep per subroutine is undefined.

The token format contains the offset of the corresponding [endloop](#) instruction in the Shader as a convenience.

The following example shows how to use the loop instruction.

syntax

```
loop
    // example of termination condition
    if_nz r0.x
        break
    endif
    ...
endloop
```

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

It (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Component-wise vector floating point less-than comparison.

It dest[.mask], [-]src0[_abs][.swizzle], [-]src1[_abs][.swizzle]

Item	Description
<i>dest</i>	[in] The address of the result of the operation.
<i>src0</i>	[in] The value to compare to <i>src1</i> .
<i>src1</i>	[in] The value to compare to <i>src0</i> .

Remarks

This instruction performs the float comparison ($src0 < src1$) for each component, and writes the result to *dest*.

If the comparison is true, then 0xFFFFFFFF is returned for that component. Otherwise 0x00000000 is returned

Denorms are flushed before comparison; original source registers are untouched.

+0 equals -0.

Comparison with NaN returns false.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes

Shader Model	Supported
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

mad (sm4 - asm)

Article • 03/08/2023 • 2 minutes to read

Component-wise multiply & add.

```
mad[_sat] dest[.mask], [-]src0[_abs][.swizzle], [-]src1[_abs][.swizzle], [-]src2[_abs][.swizzle]
```

Item	Description
<i>dest</i>	[in] The result of the operation. $dest = src0 * src1 + src2$
<i>src0</i>	[in] The multiplicand.
<i>src1</i>	[in] The multiplier.
<i>src2</i>	[in] The addend.

Remarks

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

max (sm4 - asm)

Article • 04/26/2021 • 2 minutes to read

Component-wise float maximum.

```
max[_sat] dest[.mask], [-]src0[_abs][.swizzle], [-]src1[_abs][.swizzle],
```

Item	Description
<i>dest</i>	[in] The result of the operation. $dest = src0 >= src1 ? src0 : src1$
<i>src0</i>	[in] The components to compare to <i>src1</i> .
<i>src1</i>	[in] The components to compare to <i>src0</i> .

Remarks

= is used instead of > so that if $\min(x,y) = x$ then $\max(x,y) = y$.

NaN has special handling. If one source operand is NaN, then the other source operand is returned and the choice is made per-component. If both are NaN, any NaN representation is returned.

Denorms are flushed with sign preserved before the comparison. However, the result written to *dest* may or may not be denorm flushed.

The following table shows the results obtained when executing the instruction with various classes of numbers, assuming that neither overflow or underflow occurs. F means finite real number.

src0 src1->	-inf	F	+inf	NaN
-inf	-inf	src1	+inf	-inf
F	src0	src0 or src1	+inf	src0
+inf	+inf	+inf	+inf	+inf
NaN	-inf	src1	+inf	NaN

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

min (sm4 - asm)

Article • 04/26/2021 • 2 minutes to read

Component-wise float minimum.

```
min[_sat] dest[.mask], [-]src0[_abs][.swizzle], [-]src1[_abs][.swizzle],
```

Item	Description
<i>dest</i>	[in] The result of the operation. $dest = src0 < src1 ? src0 : src1$
<i>src0</i>	[in] The components to compare to <i>src1</i> .
<i>src1</i>	[in] The components to compare to <i>src0</i> .

Remarks

= is used instead of > so that if $\min(x,y) = x$ then $\max(x,y) = y$.

NaN has special handling. If one source operand is NaN, then the other source operand is returned and the choice is made per-component. If both are NaN, any NaN representation is returned. This conforms to new IEEE 754R rules.

Denorms are flushed, with the sign preserved, before comparison. However, the result written to *dest* may or may not be denorm flushed.

The following table shows the results obtained when executing the instruction with various classes of numbers, assuming that neither overflow or underflow occurs. F means finite real number.

src0 src1->	-inf	F	+inf	NaN
-inf	-inf	-inf	-inf	-inf
F	-inf	src0 or src1	src0	src0
-inf	-inf	src1	+inf	+inf
NaN	-inf	src1	+inf	NaN

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

mov (sm4 - asm)

Article • 03/08/2023 • 2 minutes to read

Component-wise move.

```
mov[_sat] dest[.mask], [-]src0[_abs][.swizzle]
```

Item	Description
<i>dest</i>	[in] The address of the result of the operation. $dest = src0$
<i>src0</i>	[in] The components to move.

Remarks

The modifiers, other than swizzle, assume the data is floating point. The absence of modifiers just moves data without altering bits.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

movc (sm4 - asm)

Article • 03/09/2021 • 2 minutes to read

Component-wise conditional move.

```
movc[_sat] dest[.mask], src0[.swizzle], [-]src1[_abs][.swizzle], [-]src2[_abs][.swizzle],
```

Item	Description
<i>dest</i>	[in] The address of the result of the operation. If <i>src0</i> , then <i>dest</i> = <i>src1</i> else <i>dest</i> = <i>src2</i>
<i>src0</i>	[in] The components on which to test the condition.
<i>src1</i>	[in] The components to move.
<i>src2</i>	[in] The components to move.

Remarks

The following example shows how to use this instruction.

syntax

```
for each component in dest[.mask]
    if the corresponding component in src0 (POS-swizzle)
        has any bit set
    {
        copy this component (POS-swizzle) from src1 into
dest
    }
    else
    {
        copy this component (POS-swizzle) from src2 into
dest
    }
endfor
```

The modifiers on *src1* and *src2*, other than swizzle, assume the data is floating point. The absence of modifiers just moves data without altering bits.

This instruction applies to the following shader stages:

Vertex Shader

Geometry Shader

Pixel Shader

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

mul (sm4 - asm)

Article • 04/26/2021 • 2 minutes to read

Component-wise multiply.

mul[_sat] dest[.mask], [-]src0[_abs][.swizzle], [-]src1[_abs][.swizzle]

Item	Description
<i>dest</i>	[in] The result of the operation. $\text{dest} = \text{src0} * \text{src1}$
<i>src0</i>	[in] The multiplicand.
<i>src1</i>	[in] The multiplier.

Remarks

The following table shows the results obtained when executing the instruction with various classes of numbers, assuming that neither overflow or underflow occurs.

F means finite-real number.

src0	src1	-	-inf	-F	-1.0	-	-0	+0	denorm	+1.0	+F	+inf	NaN
> denorm													
-inf	+inf	+inf	+inf	+inf	Nan	Nan	Nan	Nan	-inf	-inf	-inf	-inf	Nan
-F	+inf	+F	-	+0	src0	+0	-0	-0	src0	-F	-inf	-inf	Nan
-1	+inf	-	+1.0	+0	src1	+0	-0	-0	-1.0	-	-inf	-inf	Nan
-denorm	Nan	+0	+0	+0		+0	-0	-0	-0	-0	Nan	Nan	Nan
-0	Nan	+0	+0	+0		+0	-0	-0	-0	-0	Nan	Nan	Nan
+0	iNaN	-0	-0	-0		-0	+0	+0	+0	+0	Nan	Nan	Nan
+denorm	Nan	-0	-0	-0		-0	+0	+0	+0	+0	Nan	Nan	Nan
+1.0	-inf	src1	-1.0	-0		-0	+0	+0	+1.0	src1	+inf	+inf	Nan
+F	-inf	-F	-	-0	src0	-0	+0	+0	src0	+F	+inf	+inf	Nan
+inf	-inf	-inf	-inf	Nan		Nan	Nan	Nan	+inf	+inf	+inf	+inf	Nan

src0	src1	-	-inf	-F	-1.0	-	-0	+0	denorm	+1.0	+F	+inf	NaN
>						denorm							

NaN													
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

ne (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Component-wise vector floating point not-equal comparison.

```
ne dest[.mask], [-]src0[_abs][.swizzle], [-]src1[_abs][.swizzle]
```

Item	Description
<i>dest</i>	[in] The result of the operation.
<i>src0</i>	[in] The components to compare to <i>src1</i> .
<i>src1</i>	[in] The components to compare to <i>src0</i> .

Remarks

This instruction performs the float comparison ($src0 \neq src1$) for each component, and writes the result to *dest*.

If the comparison is true, then 0xFFFFFFFF is returned for that component. Otherwise 0x0000000 is returned.

Denorms are flushed before comparison with original source registers untouched.

+0 equals -0.

Comparison with NaN returns true.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes

Shader Model	Supported
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

nop (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Do nothing.

```
nop
```

Remarks

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

not (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Bitwise not.

not dest[.mask], src0[.swizzle]

Item	Description
<i>dest</i>	[in] The address of the result of the operation.
<i>src0</i>	[in] The original components.

Remarks

This instruction performs a component-wise one's complement of each 32-bit value in *src0*. The 32-bit results are stored in *dest*.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

Shader Model 4 Assembly (DirectX HLSL)

or (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Bitwise or.

or dest[.mask], src0[.swizzle], src1[.swizzle]

Item	Description
<i>dest</i>	[in] The result of the operation.
<i>src0</i>	[in] The components to OR with <i>src1</i> .
<i>src1</i>	[in] The components to OR with <i>src0</i> .

Remarks

This instruction performs a component-wise logical OR of each pair of 32-bit values from *src0* and *src1*. The 32-bit results are placed in *dest*.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

resinfo (sm4 - asm)

Article • 03/08/2023 • 3 minutes to read

Query the dimensions of a given input resource.

```
resinfo[_uint]_rcpFloat dest[.mask], srcMipLevel.select_component, srcResource[.swizzle]
```

Item	Description
<i>dest</i>	[in] The address of the result of the operation.
<i>srcMipLevel</i>	[in] The mip level.
<i>srcResource</i>	[in] A t# or u# input texture for which the dimensions are being queried.

Remarks

srcMipLevel is read as an unsigned integer scalar so a single component selector is required for the source register, if it is not a scalar immediate value.

dest receives [width, height, depth or array size, total-mip-count], selected by the write mask.

The returned width, height and depth values are for the mip-level selected by the *srcMipLevel* parameter, and are in number of texels, independent of texel data size. For multisample resources (texture2D[Array]MS#), width and height are also returned in texels, not samples.

The total mip count returned in *dest.w* is unaffected by the *srcMipLevel* parameter.

For UAVs (u#), the number of mip levels is always 1.

All aspects of this instruction are based on the characteristics of the resource view bound at the t#/u#, not the underlying base resource.

Returned values are all floating point, unless the _uint modifier is used, in which case the returned values are all integers. If the _rcpFloat modifier is used, all returned values are floating point, and the width, height and depth are returned as reciprocals (1.0f/width, 1.0f/height, 1.0f/depth), including INF if width/height/depth are 0 from out-of-range *srcMipLevel* behavior. The _rcpFloat modifier only applies to width, height, and depth returned values and does not apply to values that are set to 0 and thus not returned, and also does not apply to array size returns.

The swizzle on *srcResource* allows the returned values to be swizzled arbitrarily before they are written to the destination.

If *srcResource* is a Texture1D, then width is returned in *dest.x*, and *dest.yz* are set to 0.

If *srcResource* is a Texture1DArray, then width is returned in *dest.x*, the array size is returned in *dest.y*, and *dest.z* is set to 0.

If *srcResource* is a Texture2D, then width and height are returned in *dest.xy*, and *dest.z* is set to 0.

If *srcResource* is a Texture2DArray, then width and height are returned in *dest.xy*, and the array size is returned in *dest.z*.

If *srcResource* is a Texture3D, then width, height and depth are returned in *dest.xyz*.

If *srcResource* is a TextureCube, then the width and height of the individual cube face dimensions are returned in *dest.xy*, and *dest.z* is set to 0.

If *srcResource* is a TextureCubeArray, then the width and height the individual cube face dimensions are returned in *dest.xy*. *dest.z* is set to an undefined value.

If the a per-resource mip clamp has been specified on *srcResource*, *resinfo* always returns the total number of mips in the view for the mip count, regardless of the clamp. However, if the dimensions of a given miplevel are requested by *resinfo* and the miplevel has been clamped off (e.g. a clamp of 2.2 means that mips 0 and 1 have been clamped off), the dimensions returned are undefined. Some implementations will return the out of bounds behavior specified for *resinfo* when the miplevel is out of range. Other implementations will return the dimensions of the mip as if it had not been clamped.

Restrictions

- *srcResource* must be a t# or u# register that is not a Buffer, but is a Texture*.
- Relative addressing of *srcResource* is not permitted.
- *srcMipLevel* must use a single component selector if it is not a scalar immediate.
- Fetching from t# or u# that has nothing bound to it returns 0 for width, height, depth or arraysize, and total-mip-count. The _rcpFloat modifier is still honored in this case, thus returning INF for the applicable returned values.
- If *srcMipLevel* is out of the range of the available number of mipmap levels in the resource, the behavior for the size return (*dest.xyz*) is identical to that of an unbound t# or u# resource. The total mip count is still returned in *dest.w* for this case.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

ret (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Return statement.

```
ret
```

Remarks

If within a subroutine, return to the instruction after the call. If not inside a subroutine, terminate program execution.

The following example shows how to use this instruction.

syntax

```
...
call 13
...
ret
label 13
...
ret
```

Restrictions

- `ret` can appear anywhere in a program, any number of times.
- If a `label` instruction appears in a Shader, it must be preceded by a `ret` command that is not nested in any flow control statements.
- If there are subroutines in a Shader, the last instruction in the Shader must be a `ret`.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

retc (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Conditional return.

retc{_z|_nz} src0.select_component

Item	Description
<i>src0</i>	[in] The register to test the condition against.

Remarks

If within a subroutine, this instruction conditionally returns to the instruction after the call. If not inside a subroutine, this instruction terminates program execution.

The following example shows how to use this instruction.

syntax

```
...
call 13
...
ret
label 13
...
retc_nz r0.x // If any bit in r0.x is nonzero, then return
retc_z r1.x // If all bits in r0.x are zero, then return.
...
ret
```

Restrictions

- **retc** can appear anywhere in a program, any number of times.
- The last instruction in a main program or subroutine cannot be a **retc_z** or **retc_nz**. Instead, the unconditional **ret** can be used.
- The 32-bit register supplied by *src0* is tested at a bit level. If any bit is nonzero, **retc_nz** will return. If all bits are zero, **retc_z** will return.

This instruction applies to the following shader stages:

Vertex Shader

Geometry Shader

Pixel Shader

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

round_ne (sm4 - asm)

Article • 04/26/2021 • 2 minutes to read

Floating-point round to integral float.

```
round_ne[_sat] dest[.mask], [-]src0[_abs][.swizzle]
```

Item	Description
<i>dest</i>	[in] The address of the results of the operation.
<i>src0</i>	[in] The components in the operation.

Remarks

This instruction performs a component-wise floating-point round of the values in *src0*, writing integral floating-point values to *dest*. **round_ne** rounds towards nearest even.

The following table shows the results obtained when executing the instruction with various classes of numbers.

F means finite-real number.

src	-inf	-F	-denorm	-0	+0	+denorm	+F	+inf	NaN
dest	-inf	-F	-0	-0	+0	+0	+F	+inf	NaN

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes

Shader Model	Supported
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

round_ni (sm4 - asm)

Article • 04/26/2021 • 2 minutes to read

Floating-point round to integral float.

```
round_ni[_sat] dest[.mask], [-]src0[_abs][.swizzle]
```

Item	Description
<i>dest</i>	[in] The address of the results of the operation.
<i>src0</i>	[in] The components in the operation.

Remarks

This instruction performs a component-wise floating-point round of the values in *src0*, writing integral floating-point values to *dest*. **round_ni** rounds toward -infinity, commonly known as floor().

The following table shows the results obtained when executing the instruction with various classes of numbers.

F means finite-real number.

<i>src</i>	-inf	-F	-denorm	-0	+0	+denorm	+F	+inf	NaN
<i>dest</i>	-inf	-F	-0	-0	+0	+0	+F	+inf	NaN

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes

Shader Model	Supported
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

round_pi (sm4 - asm)

Article • 04/26/2021 • 2 minutes to read

Floating-point round to integral float.

```
round_pi[_sat] dest[.mask], [-]src0[_abs][.swizzle]
```

Item	Description
<i>dest</i>	[in] The address of the results of the operation.
<i>src0</i>	[in] The components in the operation.

Remarks

This instruction performs a component-wise floating-point round of the values in *src0*, writing integral floating-point values to *dest*.

`round_pi` rounds towards +infinity, commonly known as `ceil()`.

The following table shows the results obtained when executing the instruction with various classes of numbers.

F means finite-real number.

src	-inf	-F	-denorm	-0	+0	+denorm	+F	+inf	NaN
dest	-inf	-F	-0	-0	+0	+0	+F	+inf	NaN

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes

Shader Model	Supported
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

round_z (sm4 - asm)

Article • 04/26/2021 • 2 minutes to read

Floating-point round to integral float.

```
round_z[_sat] dest[.mask], [-]src0[_abs][.swizzle]
```

Item	Description
<i>dest</i>	[in] The address of the results of the operation.
<i>src0</i>	[in] The components in the operation.

Remarks

This instruction performs a component-wise floating-point round of the values in *src0*, writing integral floating-point values to *dest*.

round_z rounds towards zero.

The following table shows the results obtained when executing the instruction with various classes of numbers.

src	-inf	-F	-denorm	-0	+0	+denorm	+F	+inf	NaN
dest	-inf	-F	-0	-0	+0	+0	+F	+inf	NaN

F means finite-real number.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes

Shader Model	Supported
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

rsq (sm4 - asm)

Article • 04/26/2021 • 2 minutes to read

Component-wise reciprocal square root.

```
rsq[_sat] dest[.mask], [-]src0[_abs][.swizzle]
```

Item	Description
<i>dest</i>	[in] Contains the results of the operation. $dest = 1.0f / \sqrt{src0}$
<i>src0</i>	[in] The components for the operation.

Remarks

The maximum relative error is 2-21.

The following table shows the results obtained when executing the instruction with various classes of numbers, assuming that neither overflow or underflow occurs.

F means finite-real number.

src	-inf	-F	-denorm	-0	+0	+denorm	+F	+inf	NaN
dest	NaN	NaN	-inf	-inf	+inf	+inf	+F	+0	NaN

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes

Shader Model	Supported
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

sample (sm4 - asm)

Article • 03/09/2021 • 4 minutes to read

Samples data from the specified Element/texture using the specified address and the filtering mode identified by the given sampler.

```
sample[_aoffimmi(u,v,w)] dest[.mask], srcAddress[.swizzle], srcResource[.swizzle],  
srcSampler
```

Item	Description
<i>dest</i>	[in] The address of the result of the operation.
<i>srcAddress</i>	[in] A set of texture coordinates. For more information, see the Remarks section.
<i>srcResource</i>	[in] A texture register. For more information, see the Remarks section.
<i>srcSampler</i>	[in] A sampler register. For more information, see the Remarks section.

Remarks

The source data may come from any Resource Type, other than Buffers.

srcAddress provides the set of texture coordinates needed to perform the sample, as floating point values referencing normalized space in the texture. Address wrapping modes (wrap/mirror/clamp/border etc.) are applied for texture coordinates outside [0...1] range, taken from the sampler state (*s#*), and applied after any address offset is applied to texture coordinates.

srcResource is a texture register (*t#*). This is simply a placeholder for a texture, including the return data type of the resource being sampled. All of this information is declared in the Shader preamble. The actual resource to be sampled is bound to the Shader externally at slot # (for *t#*).

srcSampler is a sampler register (*s*). This is simply a placeholder for a collection of filtering controls such as point vs. linear, mipmapping and address wrapping controls.

The set of information required for the hardware to perform sampling is split into two orthogonal pieces. First, the texture register provides source data type information including, for example, information about whether the texture contains SRGB data. It also references the actual memory being sampled. Second, the sampler register defines the filtering mode to apply.

Array Resources

For Texture1D Arrays, the *srcAddress* g component (POS-swizzle) selects which Array Slice to fetch from. This is always treated as a scaled float value, rather than the normalized space for standard texture coordinates, and a round-to-nearest even is applied on the value, followed by a clamp to the available BufferArray range.

For Texture2D Arrays, the *srcAddress* b component (POS-swizzle) selects which Array Slice to fetch from, otherwise using the same semantics described for Texture1D Arrays .

Address Offset

The optional [_aoffimmi(u,v,w)] suffix (address offset by immediate integer) indicates that the texture coordinates for the sample are to be offset by a set of provided immediate texel space integer constant values. The literal values are a set of 4 bit 2's complement numbers, having integer range [-8,7]. This modifier is defined for all Resources, including Texture1D/2D Arrays and Texture3D, but it is undefined for TextureCube.

Hardware can take advantage of immediate knowledge that a traversal over some footprint of texels about a common location is being performed by a set of sample instructions. This can be conveyed using _aoffimmi(u,v,w).

The offsets are added to the texture coordinates, in texel space, relative to each miplevel being accessed. So even though texture coordinates are provided as normalized float values, the offset applies a texel-space integer offset.

Address offsets are not applied along the array axis of Texture1D/2D Arrays.

_aoffimmi v,w components are ignored for Texture1Ds.

_aoffimmi w component is ignored for Texture2Ds.

Address wrapping modes (wrap/mirror/clamp/border etc.) from the sampler state (s#) are applied after any address offset is applied to texture coordinates.

Return Type Control

The data format returned by the sample to the destination register is determined by the resource format (DXGI_FORMAT*) bound to the *srcResource* parameter (t#). For example, if the specified t# was bound with a resource with format DXGI_FORMAT_A8B8G8R8_UNORM_SRGB, then the sampling operation will convert

sampled texels from gamma 2.0 to 1.0, apply filtering, and the result will be written to the destination register as floating point values in the range [0..1].

Returned values are 4-vectors (with format-specific defaults for components not present in the format). The swizzle on *srcResource* determines how to swizzle the 4-component result coming back from the texture sample/filter, after which *.mask* on *dest* determines which components in *dest* get updated.

When **sample** reads a 32-bit float value into a 32-bit register, with point sampling (no filtering), it may or may not flush denormal values, but otherwise numbers are unmodified. If the uncertainty with point sampling denormal values is an issue for an application, use the **ld** instruction, which guarantees that 32-bit float values are read unmodified.

LOD Calculation

For details on how derivatives are calculated in the process of determining LOD for filtering, see [deriv_rtx](#) and [deriv_rty](#). The **sample** instruction implicitly computes derivatives on the texture coordinates using the same definition that the **deriv** Shader instructions use. This does not apply to [sample_l](#) or [sample_d](#) instructions. For those instructions, LOD or derivatives are provided directly by the application.

For the **sample** instruction, implementations can choose to share the same LOD calculation across all 4 pixels in a 2x2 stamp (but no larger area), or perform per-pixel LOD calculations.

Miscellaneous Details

For Buffer & Texture1D, *srcAddress .gba* components (POS-swizzle) are ignored. For Texture1D Arrays, *srcAddress .ba* components (POS-swizzle) are ignored. For Texture2Ds, *srcAddress .a* component (POS-swizzle) is ignored.

Fetching from an input slot that has nothing bound to it returns 0 for all components.

Restrictions

- *srcResource* must be a t# register. *srcResource* cannot be a ConstantBuffer, which cannot be bound to t# registers.
- *srcSampler* must be an s# register.
- Relative addressing on *srcResource* or *srcSampler* is not permitted.
- *srcAddress* must be a temp (r#/x#), constantBuffer (cb#), input (v#) register or immediate value(s).

- *dest* must be a temp (r#/x#) or output (o*#) register.
- `_aoffimmi(u,v,w)` is not permitted for TextureCubes.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
		x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

sample_b (sm4 - asm)

Article • 03/09/2021 • 2 minutes to read

Samples data from the specified Element/texture using the specified address and the filtering mode identified by the given sampler.

```
sample_b[_aoffimmi(u,v,w)] dest[.mask], srcAddress[.swizzle], srcResource[.swizzle],  
srcSampler, srcLODBias.select_component
```

Item	Description
<i>dest</i>	[in] The address of the result of the operation.
<i>srcAddress</i>	[in] A set of texture coordinates. For more information see the sample instruction.
<i>srcResource</i>	[in] A texture register. For more information see the sample instruction.
<i>srcSampler</i>	[in] A sampler register. For more information see the sample instruction.
<i>srcLODBias</i>	[in] See the Remarks section for information about this parameter.

Remarks

The source data may come from any Resource Type, other than Buffers. An additional bias is applied to the level of detail computed as part of the instruction execution.

This instruction behaves like the [sample](#) instruction with the addition of applying the specified *srcLODBias* value to the level of detail value computed as part of the instruction execution prior to selecting the mip map(s). The *srcLODBias* value is added to the computed LOD on a per-pixel basis, along with the sampler MipLODBias value, prior to the clamp to MinLOD and MaxLOD.

Restrictions

- `sample_b` inherits the same restrictions as the [sample](#) instruction, plus additional restrictions for its additional parameter.
- The range of *srcLODBias* is (-16.0f to 15.99f); values outside of this range will produce undefined results.
- *srcLODBias* must use a single component selector if it is not a scalar immediate.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
		x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

sample_c (sm4 - asm)

Article • 03/08/2023 • 2 minutes to read

Performs a comparison filter.

```
sample_c[_aoffimmi(u,v,w)] dest[.mask], srcAddress[.swizzle], srcResource.r, srcSampler,  
srcReferenceValue
```

Item	Description
<i>dest</i>	[in] The address of the results of the operation.
<i>srcAddress</i>	[in] A set of texture coordinates. For more information see the sample instruction.
<i>srcResource</i>	[in] A texture register. For more information see the sample instruction. Must be .r swizzle.
<i>srcSampler</i>	[in] A sampler register. For more information see the sample instruction.
<i>srcReferenceValue</i>	[in] A register with a single component selected, which is used in the comparison.

Remarks

The primary purpose for this instruction is to provide a building-block for Percentage-Closer Depth filtering. The "c" in **sample_c** stands for Comparison.

The operands to **sample_c** are identical to the [sample](#) instruction, except that there is an additional float32 source operand, *srcReferenceValue*, which must be a register with single-component selected, or a scalar literal.

The *srcResource* parameter must have a .r (red) swizzle. **sample_c** operates exclusively on the red component, and returns a single value. The .r swizzle on *srcResource* indicates that the scalar result is replicated to all components.

When a Depth Buffer is set as an input texture, the depth value shows up in the red component.

If this instruction is used with a Resource that is not a Texture1D/2D/2DArray/Cube/CubeArray, it produces undefined results.

When this instruction is executed, the sampling hardware uses the current Sampler's ComparisonFunction to compare *srcReferenceValue* against the Red component value

for the source Resource at each filter "tap" location (texel) that the currently configured texture filter covers, based on the provided coordinates in *srcAddress*.

The comparison occurs after *srcReferenceValue* has been quantized to the precision of the texture format, in exactly the same way that z is quantized to depth buffer precision before Depth Comparison at the Output Merger visibility test. This includes a clamp to the format range (e.g. [0..1] for a UNORM format).

The source texel's Red component is compared against the quantized *srcReferenceValue*. For texels that fall off the Resource, the Red component value is determined by applying the Address Modes (and BorderColorR if in Border mode) from the Sampler. The comparison honors all D3D11 floating point comparison rules, in the case the texture format is floating point.

Each comparison that passes returns 1.0f as the Red component value for the texel, and each comparison that fails returns 0.0f as the Red value for the texture. Filtering then occurs exactly as specified by the Sampler states, operating only in the Red component, and returning a single scalar filter result back to the Shader, replicated to all masked *dest* components.

The use of **sample_c** is orthogonal to all other general purpose filtering controls. **sample_c** works seamlessly with the other general purpose filter modes. **sample_c** changes the behavior of the general purpose filters such that the values being filtered all become 1.0f or 0.0f due to comparison results.

Fetching from an input slot that has nothing bound to it returns 0 for all components.

For more information, see the [sample](#) instruction.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
		x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes

Shader Model	Supported
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

sample_c_lz (sm4 - asm)

Article • 03/08/2023 • 2 minutes to read

Performs a comparison filter. This instruction behaves like [sample_c](#), except LOD is 0, and derivatives are ignored.

```
sample_c_lz[_aoffimmi(u,v,w)] dest[.mask], srcAddress[.swizzle], srcResource.r, srcSampler,  
srcReferenceValue
```

Item	Description
<i>dest</i>	[in] The address of the results of the operation.
<i>srcAddress</i>	[in] A set of texture coordinates. For more information see the sample instruction.
<i>srcResource</i>	[in] A texture register. For more information see the sample instruction. Must be .r swizzle.
<i>srcSampler</i>	[in] A sampler register. For more information see the sample instruction.
<i>srcReferenceValue</i>	[in] A register with a single component selected, which is used in the comparison.

Remarks

The "lz" stands for level-zero. Because derivatives are ignored, this instruction is available in shaders other than the Pixel Shader.

If this instruction is used with a mipmapped texture, LOD 0 gets sampled, unless the sampler has an LOD clamp which places the LOD somewhere else, or if there is an LOD Bias, which would simply bias starting from 0. Because derivatives are ignored, anisotropic filtering behaves as isotropic filtering.

In Pixel Shaders, this instruction can be used inside varying flow control when the texture coordinates are derived in the shader, unlike [sample_c](#).

Fetching from an input slot that has nothing bound to it returns 0 for all components.

This instruction is available in all shaders, not just the Pixel Shader, for consistency.

Vertex Shader	Geometry Shader	Pixel Shader
X	X	X

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

sample_d (sm4 - asm)

Article • 03/08/2023 • 2 minutes to read

Samples data from the specified Element/texture using the specified address and the filtering mode identified by the given sampler.

```
sample_d[_aoffimmi(u,v,w)] dest[.mask], srcAddress[.swizzle], srcResource[.swizzle],  
srcSampler, srcXDerivatives[.swizzle], srcYDerivatives[.swizzle]
```

Item	Description
<i>dest</i>	[in] The address of the results of the operation.
<i>srcAddress</i>	[in] A set of texture coordinates. For more information see the sample instruction.
<i>srcResource</i>	[in] A texture register. For more information see the sample instruction.
<i>srcSampler</i>	[in] A sampler register. For more information see the sample instruction.
<i>srcXDerivatives</i>	[in] The derivatives for the source address in the x direction. For more information, see the Remarks section.
<i>srcYDerivatives</i>	[in] The derivatives for the source address in the y direction. For more information, see the Remarks section.

Remarks

This instruction behaves like the [sample](#) instruction, except that derivatives for the source address in the x direction and the y direction are provided by extra parameters, *srcXDerivatives* and *srcYDerivatives*, respectively. These derivatives are in normalized texture coordinate space.

The r, g and b components of *srcXDerivatives* (POS-swizzle) provide du/dx, dv/dx and dw/dx. The 'a' component (POS-swizzle) is ignored.

The r, g and b components of *srcYDerivatives* (POS-swizzle) provide du/dy, dv/dy and dw/dy. The 'a' component (POS-swizzle) is ignored.

Unlike the [sample](#) instruction, which is permitted to share a single LOD calculation across a 2x2 stamp, **sample_d** must calculate LOD completely independently, per-pixel when used in the Pixel Shader.

If the derivative inputs to `sample_d` came from derivative calculation instructions in the Pixel Shader and the values include INF/NaN, the behavior of `sample_d` may not match the `sample` instruction, which implicitly computes the derivative. The INF/NaN values may affect the LOD calculation differently.

Fetching from an input slot that has nothing bound to it returns 0 for all components.

Restrictions

- `sample_d` inherits the same restrictions as the `sample` instruction, plus additional an restriction below for its additional parameters.
- `srcXDerivatives` and `srcYDerivatives` must be temp (`r#/x#`), constantBuffer (`cb#`), input (`v#`) registers or immediate value(s).

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
X	X	X

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

sample_I (sm4 - asm)

Article • 03/09/2021 • 2 minutes to read

Samples data from the specified Element/texture using the specified address and the filtering mode identified by the given sampler.

```
sample_I[_aoffimmi(u,v,w)] dest[.mask], srcAddress[.swizzle], srcResource[.swizzle],  
srcSampler, srcLOD.select_component
```

Item	Description
<i>dest</i>	[in] The address of the results of the operation.
<i>srcAddress</i>	[in] A set of texture coordinates. For more information see the sample instruction.
<i>srcResource</i>	[in] A texture register. For more information see the sample instruction.
<i>srcSampler</i>	[in] A sampler register. For more information see the sample instruction.
<i>srcLOD</i>	[in] The LOD.

Remarks

This instruction is identical to [sample](#), except that LOD is provided directly by the application as a scalar value, representing no anisotropy. This instruction is available in all programmable Shader stages.

sample_I samples the texture using *srcLOD* to be the LOD. If the LOD value is ≤ 0 , the zero'th (biggest map) is chosen, with the magnify filter applied (if applicable based on the filter mode). Because *srcLOD* is a floating point value, the fractional value is used to interpolate between two mip levels, if the minify filter is LINEAR or with anisotropic filtering.

sample_I ignores address derivatives, so filtering behavior is purely isotropic. Because derivatives are ignored, anisotropic filtering behaves as isotropic filtering.

Sampler states MIPLODBIAS and MAX/MINMIPLEVEL are honored.

When used in the Pixel Shader, **sample_I** implies that the choice of LOD is per-pixel, with no effect from neighboring pixels, for example in the same 2x2 stamp.

Fetching from an input slot that has nothing bound to it returns 0 for all components.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
X	X	X

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

sampleinfo (sm4.1 - asm)

Article • 03/08/2023 • 2 minutes to read

Queries the number of samples in a given shader resource view or in the rasterizer.

```
sampleinfo[_uint] dest[.mask], srcResource[.swizzle]
```

Item	Description
<i>dest</i>	[in] The address of the results of the operation.
<i>srcResource</i>	[in] The shader resource.

Remarks

This instruction returns the number of samples for the given resource or the rasterizer. It is valid only for resources that can be loaded using [Id2dms](#) unless the rasterizer is specified as *srcResource*. *srcResource* could be a t# register (a shader resource view) or a rasterizer register.

The instruction computes the vector (SampleCount,0,0,0).

The swizzle on *srcResource* allows the returned values to be swizzled arbitrarily before they are written to the destination. The returned value is floating point, unless the _uint modifier is used, in which case the returned value is integer. If there is no resource bound to the specified slot, 0 is returned.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
X	X	X

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes

Shader Model	Supported
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

samplepos (sm4.1 - asm)

Article • 03/08/2023 • 2 minutes to read

Queries the position of a sample in a given shader resource view or in the rasterizer.

samplepos dest[.mask], srcResource[.swizzle], sampleIndex

Item	Description
<i>dest</i>	[in] The address of the results of the operation.
<i>srcResource</i>	[in] The shader resource.
<i>sampleIndex</i>	[in] The index of the sample (scalar operand).

Remarks

This instruction returns the 2D sample position of sample *sampleIndex* for the given resource. It is valid only for resources that can be loaded using [Id2dms](#) unless the rasterizer is specified as *srcResource*.

srcResource can be a t# register (a shader resource view) or a rasterizer register.

The instruction computes the floating point vector (Xposition, Yposition, 0, 0).

The swizzle on *srcResource* allows the returned values to be swizzled arbitrarily before they are written to the destination. The sample position is relative to the pixel's center, based on the Pixel Coordinate System.

If *sampleIndex* is out of bounds a zero vector is returned. If there is no resource bound to the specified slot, 0 is returned.

samplepos can be used for things like custom resolves in shader code.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
		x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

sincos (sm4 - asm)

Article • 04/26/2021 • 2 minutes to read

Component-wise sin(theta) and cos(theta) for theta in radians.

```
sincos[_sat] destSIN[.mask], destCOS[.mask], [-]src0[_abs][.swizzle]
```

Item	Description
<i>destSIN</i>	[in] The address of sin(<i>src0</i>), computed per component.
<i>destCOS</i>	[in] The address of cos(<i>src0</i>), computed per component.
<i>src0</i>	[in] The components for which to compute sin and cos.

Remarks

If the result is not needed, you can specify *destSIN* and *destCOS* as NULL instead of specifying a register.

Theta values can be any IEEE 32-bit floating point values.

The maximum absolute error is 0.0008 in the interval from -100*Pi to +100*Pi.

The following table shows the results obtained when executing the instruction with various classes of numbers.

F means finite-real number.

src	-inf	-F	-denorm	-0	+0	+denorm	+F	+inf	NaN
destSIN	NaN	[-1 to +1]	-0	-0	+0	+0	[-1 to +1]	NaN	NaN
destCOS	NaN	[-1 to +1]	+1	+1	+1	+1	[-1 to +1]	NaN	NaN

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

sqrt (sm4 - asm)

Article • 04/26/2021 • 2 minutes to read

Component-wise square root.

```
sqrt[_sat] dest[.mask], [-]src0[_abs][.swizzle]
```

Item	Description
<i>dest</i>	[in] The result of the operation. $dest = \sqrt{src0}$
<i>src0</i>	[in] The components for which to take the square root.

Remarks

Precision is 1 ulp.

The following table shows the results obtained when executing the instruction with various classes of numbers, assuming that neither overflow or underflow occurs.

F means finite-real number.

	-inf	-F	-denorm	-0	+0	+denorm	+F	+inf	NaN
dest	NaN	NaN	-0	-0	+0	+0	+F	+inf	NaN

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes

Shader Model	Supported
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

switch (sm4 - asm)

Article • 03/08/2023 • 2 minutes to read

Transfers control to a different statement block within the switch body depending on the value of a selector.

```
switch src0.select_component
```

Item	Description
src0	[in] The selector for the switch statement.

Remarks

A **switch/endswitch** construct behaves exactly as a **switch** construct in the C language, with the following exception: for D3D11 **case/default** statements that fall through to the next **case/default** without a **break** cannot have any code in them. It is permitted for multiple **case** statements, including **default**, to appear sequentially, sharing the same code block.

The condition must be a 32-bit register component or immediate quantity. The equality comparison is bitwise (integer).

As with any Shader instruction in the D3D11, hardware may or may not implement the **switch** construct directly.

Switch statements can be nested. Each **switch** block counts as 1 level against the flow control nesting depth limit of 64 per subroutine and main, independent of the number of **case** statements. The HLSL compiler will not generate subroutines that exceed this limit. Behavior of control flow instructions beyond 64 levels deep per subroutine is undefined.

The following example shows how to use this instruction.

syntax

```
...
switch r0.x
default: // falling through
case 3
    switch r1.x
    case 4
    ...

```

```
    break
    case 5
    ...
    break
endswitch
break
case 0
    break
endswitch
```

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

udiv (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Unsigned integer divide.

udiv destQUOT[.mask], destREM[.mask], src0[.swizzle], src1[.swizzle]

Item	Description
<i>destQUOT</i>	[in] The address of the resulting quotient.
<i>destREM</i>	[in] The address of the resulting remainder.
<i>src0</i>	[in] The components to be divided by <i>src1</i> .
<i>src1</i>	[in] The components by whch to divide <i>src0</i> .

Remarks

This instruction performs a component-wise unsigned divide of the 32-bit operand *src0* by the 32-bit operand *src1*. The results of the divides are the 32-bit quotients placed in *destQUOT* and 32-bit remainders placed in *destREM*.

Divide by zero returns 0xffffffff for both quotient and remainder.

You can specify either *destQUOT* or *destREM* as NULL instead of specifying a register, if the quotient or remainder are not needed.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes

Shader Model	Supported
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

uge (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Component-wise vector unsigned integer greater-than-or-equal comparison.

uge dest[.mask], src0[.swizzle], src1[.swizzle]

Item	Description
<i>dest</i>	[in] The address of the result of the operation.
<i>src0</i>	[in] The components to compare to <i>src1</i> .
<i>src1</i>	[in] The components to compare to <i>src0</i> .

Remarks

This instruction performs the unsigned integer comparison ($src0 \geq src1$) for each component, and writes the result to *dest*.

If the comparison is true, then 0xFFFFFFFF is returned for that component. Otherwise 0x0000000 is returned.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no

Shader Model	Supported
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

ult (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Component-wise vector unsigned integer less-than comparison.

ult dest[.mask], src0[.swizzle], src1[.swizzle]

Item	Description
<i>dest</i>	[in] The address of the result of the operation.
<i>src0</i>	[in] The value to compare to <i>src1</i> .
<i>src1</i>	[in] The value to compare to <i>src1</i> .

Remarks

Performs the unsigned integer comparison ($src0 < src1$) for each component, and writes the result to *dest*.

If the comparison is true, this instruction returns 0xFFFFFFFF for that component. Otherwise it returns 0x00000000.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no

Shader Model	Supported
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

umad (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Unsigned integer multiply and add.

umad dest[.mask], src0[.swizzle], src1[.swizzle], src2[.swizzle]

Item	Description
<i>dest</i>	[in] The address of the result of the operation.
<i>src0</i>	[in] The value to multiply with <i>src1</i> .
<i>src1</i>	[in] The value to multiply with <i>src1</i> .
<i>src2</i>	[in] The value to add to the product of <i>src0</i> and <i>src1</i> .

Remarks

Component-wise [umul](#) of 32-bit operands *src0* and *src1* unsigned, keeping the low 32-bits, per component, of the result. This instruction then performs an [iadd](#) of *src2*, producing the correct low 32-bit (per component) result. The 32-bit results are placed in *dest*.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no

Shader Model	Supported
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

umax (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Component-wise unsigned integer maximum.

```
umax dest[.mask], src0[.swizzle], src1[.swizzle],
```

Item	Description
<i>dest</i>	[in] The address of the result of the operation. $dest = src0 > src1 ? src0 : src1$
<i>src0</i>	[in] The value to compare to <i>src1</i> .
<i>src1</i>	[in] The value to compare to <i>src0</i> .

Remarks

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

Shader Model 4 Assembly (DirectX HLSL)

umin (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Component-wise unsigned integer minimum.

```
umin dest[.mask], src0[.swizzle], src1[.swizzle],
```

Item	Description
<i>dest</i>	[in] The address of the result of the operation. $dest = src0 < src1 ? src0 : src1$
<i>src0</i>	[in] The value to compare to <i>src1</i> .
<i>src1</i>	[in] The value to compare to <i>src0</i> .

Remarks

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

Shader Model 4 Assembly (DirectX HLSL)

umul (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Unsigned integer multiply.

umul destHI[.mask], destLO[.mask], src0[.swizzle], src1[.swizzle]

Item	Description
<i>destHI</i>	[in] The high 32 bits of the result, per component.
<i>destLO</i>	[in] The low 32 bits of the result, per component.
<i>src0</i>	[in] The components by which to multiply <i>src1</i> .
<i>src1</i>	[in] The components by which to multiply <i>src0</i> .

Remarks

This instruction performs a component-wise multiply of unsigned 32-bit operands *src0* and *src1*, producing the correct full 64-bit result per component. The low 32 bits per component are placed in *destLO*. The high 32 bits per component are placed in *destHI*.

You can specify either *destHI* or *destLO* as NULL instead of specifying a register if the high or low 32 bits of the 64-bit result are not needed.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes

Shader Model	Supported
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

ushr (sm4 - asm)

Article • 03/09/2021 • 2 minutes to read

Shift right.

ushr dest[.mask], src0[.swizzle], src1.select_component

Item	Description
<i>dest</i>	[in] The address of the result of the operation.
<i>src0</i>	[in] The components to shift.
<i>src1</i>	[in] The amount to shift <i>src0</i> .

Remarks

This instruction performs a component-wise shift of each 32-bit value in *src0* right by an unsigned integer bit count provided by the LSB 5 bits (0-31 range) in *src1.select_component*, inserting 0. The 32-bit per component result is placed in *dest*. The count is a scalar value applied to all components.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no

Shader Model	Supported
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

utof (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Unsigned integer to floating point conversion.

utof dest[.mask], src0[.swizzle]

Item	Description
<i>dest</i>	[in] The address of the result of the operation.
<i>src0</i>	[in] The components to convert.

Remarks

src0 must contain an unsigned 32-bit integer 4-tuple. After the instruction executes, *dest* will contain a floating-point 4-tuple. The conversion is performed per-component.

When an integer input value is too large to be represented exactly in the floating point format, round to nearest even mode.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

xor (sm4 - asm)

Article • 11/20/2019 • 2 minutes to read

Bitwise xor.

xor dest[.mask], src0[.swizzle], src1[.swizzle]

Item	Description
<i>dest</i>	[in] The result of the operation.
<i>src0</i>	[in] The components to XOR with <i>src1</i> .
<i>src1</i>	[in] The components to XOR with <i>src0</i> .

Remarks

This instruction performs a component-wise logical XOR of each pair of 32-bit values from *src0* and *src1*. 32-bit results are placed in *dest*.

This instruction applies to the following shader stages:

Vertex Shader	Geometry Shader	Pixel Shader
x	x	x

Minimum Shader Model

This function is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	yes
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 4 Assembly \(DirectX HLSL\)](#)

Shader Model 5 Assembly

Article • 03/08/2023 • 2 minutes to read

This section describes the instructions supported by Shader Model 5.

Instruction Modifiers

add
and
atomic_and
atomic_cmp_store
atomic_iadd
atomic_imax
atomic_imin
atomic_or
atomic_umin
atomic_xor
bfi
bfrev
break
breakc
bufinfo
call
callc
case
continue
continuec
countbits
cut
cut_stream
dadd
dcl_constantBuffer
dcl_function_body
dcl_function_table
dcl_globalFlags
dcl_hs_fork_phase_instance_count
dcl_hs_join_phase_instance_count
dcl_hs_max_tessfactor
dcl_immediateConstantBuffer
dcl_indexableTemp

dcl_indexRange
dcl_input
dcl_input vForkInstanceID
dcl_input vGSInstanceID
dcl_input vJoinInstanceID
dcl_input vOutputControlPointID
dcl_input vPrim
dcl_input vThread
dcl_input_control_point_count
dcl_input_sv
dcl_inputPrimitive
dcl_interface
dcl_interface_dynamicindexed
dcl_maxOutputVertexCount
dcl_output
dcl_output oDepth
dcl_output oMask
dcl_output_control_point_count
dcl_output_sgv
dcl_output_siv
dcl_outputTopology
dcl_resource
dcl_resource raw
dcl_resource structured
dcl_sampler
dcl_stream
dcl_temps
dcl_tessellator_domain
dcl_tessellator_output_primitive
dcl_tessellator_partitioning
dcl_tgsm_raw
dcl_tgsm_structured
dcl_thread_group
dcl_uav_raw
dcl_uav_structured
dcl_uav_typed
ddiv
default
deq
deriv_rtx_coarse

deriv_rtx_fine
deriv_rty_coarse
deriv_rty_fine
dfma
dge
discard
div
dlt
dmax
dmin
dmov
dmovc
dmul
dne
dp2
dp3
dp4
drccp
else
emit
emit_stream
emitThenCut
emitThenCut_stream
endif
endloop
endswitch
eq
exp
f16tof32
f32tof16
fcall
firstbit
frc
ftod
ftoi
ftou
gather4
gather4_c
gather4_po
gather4_po_c

ge
hs_control_point_phase
hs_decls
hs_fork_phase
hs_join_phase
iadd
ibfe
ieq
if
ige
ilt
imad
imin
imm_atomic_alloc
imm_atomic_and
imm_atomic_cmp_exch
imm_atomic_consume
imm_atomic_exch
imm_atomic_iadd
imm_atomic_imax
imm_atomic_imin
imm_atomic_or
imm_atomic_umin
imm_atomic_xor
imul
ine
ineg
ishl
ishr
itof
label
ld
ld_raw
ld_structured
ld_uav_typed
ld2dms
lod
log
loop

lt
mad
max
min
mov
movc
mul
ne
nop
not
or
rcp
resinfo
ret
retc
round_ne
round_ni
round_pi
round_z
rsq
sample
sample_b
sample_c
sample_c_lz
sample_d
sample_l
sampleinfo
samplepos
sincos
sqrt
store_raw
store_structured
store_uav_typed
swapc
switch
sync
uaddc
ubfe
udiv
uge

[ult](#)
[umad](#)
[umax](#)
[umin](#)
[umul](#)
[ushr](#)
[usubb](#)
[utof](#)
[xor](#)

Related topics

[Asm Shader Reference](#)

Shader Model 5 Instruction Modifiers

Article • 08/23/2019 • 2 minutes to read

Instruction modifiers affect the result of the instruction before it is written to the register. Shader Model 5 supports the following instruction modifiers.

Source Operand Modifiers

- [Absolute Value](#)
- [Negate](#)

Instruction Result Modifiers

- [Precise](#)
- [Saturate](#)

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

Precise

Article • 11/20/2019 • 2 minutes to read

Per-instruction disabling of arithmetic refactoring.

precise (component mask)

This modifier requires the global shader flag "REFACTORING_ALLOWED". When REFACTORING_ALLOWED is present, individual component results of individual instructions can be forced to remain precise or not-refactorable by compilers or drivers. If components of a [mad](#) instruction are tagged as **precise**, the hardware must execute a **mad** instruction or the exact equivalent, and it cannot split it into a multiply followed by an add. Conversely, a multiply followed by an add, where either or both are flagged as **precise**, cannot be merged into a fused **mad**.

If REFACTORING_ALLOWED has not been specified, the **precise** modifier is not allowed. It is not needed because everything is precise. The **precise** modifier affects any operation, not just arithmetic.

Minimum Shader Model

This modifier is supported in the following shader models.

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	yes
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

Shader Model 5 Instruction Modifiers

atomic_and (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Atomic bitwise AND to memory.

atomic_and dest, dstAddress[.swizzle], src0[.select_component]

Item	Description
<i>dest</i>	[in] The components to AND with <i>src0</i> . This value must be an unordered access view (UAV) (u#). In the compute shader it can also be thread group shared memory (g#).
<i>dstAddress</i>	[in] The memory address.
<i>src0</i>	[in] The components to AND with <i>dest</i> .

Remarks

This operation performs a single component 32-bit bitwise AND of operand *src0* into *dest* at 32-bit per component address *dstAddress*, performed atomically.

The number of components taken from the address is determined by the dimensionality of *dest* u# or g#.

If *dest* is a u#, it can be declared as raw, typed or structured. If typed, it must be declared as UINT/SINT with the bound resource format being R32_UINT/_SINT.

If *dest* is g#, it must be declared as raw or structured.

Nothing is returned to the shader.

If the shader invocation is inactive, for example if the pixel has been discarded earlier in its execution, or a pixel/sample invocation only exists to serve as a helper to a real pixel/sample for derivatives, this instruction does not alter the *dest* memory at all (silently).

Out of bounds addressing on u# causes nothing to be written to memory, except if the u# is structured, and byte offset into the struct (second component of the address) is causing the out of bounds access, then the entire contents of the UAV become undefined.

Out of bounds addressing on g# (the bounds of that particular g#, as opposed to all shared memory) causes the entire contents of all shared memory to become undefined.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

Because UAVs are available at all shader stages for Direct3D 11.1, this instruction applies to all shader stages for the Direct3D 11.1 runtime, which is available starting with Windows 8.

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

atomic_cmp_store (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Atomic compare and write to memory.

```
atomic_cmp_store dest, dstAddress[.swizzle], src0[.select_component],  
src1[.select_component]
```

Item	Description
<i>dest</i>	[in] The components to compare with <i>src0</i> . This value must be an unordered access view (UAV) (u#). In the compute shader it can also be thread group shared memory (g#).
<i>dstAddress</i>	[in] The memory address.
<i>src0</i>	[in] The 32-bit value to compare with <i>dest</i> .
<i>src1</i>	[in] The value to write to memory if the compared values are identical.

Remarks

This instruction performs a single component 32-bit value compare of operand *src0* with *dest* at 32-bit per component address *dstAddress*.

If the compared values are identical, the single-component 32-bit value in *src1* is written to destination memory. Otherwise, the destination is not changed.

The entire compare and write operation is performed atomically.

If *dest* is a u#, it can be declared as raw, typed or structured. If typed, it must be declared as UINT/SINT with the bound resource format being R32_UINT/_SINT.

If *dest* is g#, it must be declared as raw or structured.

The number of components taken from the address is determined by the dimensionality of *dest* u# or g#.

Nothing is returned to the shader.

If the shader invocation is inactive, for example if the pixel has been discarded earlier in its execution, or a pixel/sample instruction does not alter the *dest* memory at all (silently).

Out of bounds addressing on u# causes nothing to be written to memory, except if the u# is structured, and byte offset into the struct (second component of the address) is causing the out of bounds access, then the entire contents of the UAV become undefined.

Out of bounds addressing on g# (the bounds of that particular g#, as opposed to all shared memory) causes the entire contents of all shared memory to become undefined.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

Because UAVs are available at all shader stages for Direct3D 11.1, this instruction applies to all shader stages for the Direct3D 11.1 runtime, which is available starting with Windows 8.

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

atomic_iadd (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Atomic integer add to memory.

atomic_iadd dest, dstAddress[.swizzle], src0[.select_component]

Item	Description
<i>dest</i>	[in] The components to add with <i>src0</i> . This value must be an unordered access view (UAV) (u#). In the compute shader it can also be thread group shared memory (g#).
<i>dstAddress</i>	[in] The memory address.
<i>src0</i>	[in] The components to add to <i>dest</i> .

Remarks

This instruction performs a single component 32-bit integer add of operand *src0* into *dest* at 32-bit per component address *dstAddress*, performed atomically. This instruction is insensitive to sign.

The number of components taken from the address is determined by the dimensionality of *dest* u# or g#.

If *dest* is a u#, it can be declared as raw, typed or structured. If typed, it must be declared as UINT/SINT with the bound resource format being R32_UINT/_SINT.

If *dest* is g#, it must be declared as raw or structured.

Nothing is returned to the shader.

If the shader invocation is inactive, for example if the pixel has been discarded earlier in its execution, or if a pixel/sample invocation only exists to serve as a helper to a real pixel/sample for derivatives, this instruction does not alter the *dest* memory at all (silently).

Out of bounds addressing on u# causes nothing to be written to memory, except if the u# is structured, and byte offset into the struct (second component of the address) is causing the out of bounds access, then the entire contents of the UAV become undefined.

Out of bounds addressing on g# (the bounds of that particular g#, as opposed to all shared memory) causes the entire contents of all shared memory to become undefined.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

Because UAVs are available at all shader stages for Direct3D 11.1, this instruction applies to all shader stages for the Direct3D 11.1 runtime, which is available starting with Windows 8.

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

atomic_imax (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Atomic signed integer maximum to memory.

atomic_imax dest, dstAddress[.swizzle], src0[.select_component]

Item	Description
<i>dest</i>	[in] The components to compare with <i>src0</i> . This value must be an unordered access view (UAV) (u#). In the compute shader it can also be thread group shared memory (g#).
<i>dstAddress</i>	[in] The memory address.
<i>src0</i>	[in] The components to compare to <i>dest</i> .

Remarks

This operation performs a single component 32-bit signed maximum of operand *src0* into *dest* at 32-bit per component address *dstAddress*, performed atomically.

The number of components taken from the address is determined by the dimensionality of *dest* u# or g#.

If *dest* is a u#, it can be declared as raw, typed or structured. If typed, it must be declared as UINT/SINT with the bound resource format being R32_UINT/_SINT.

If *dest* is g#, it must be declared as raw or structured.

Nothing is returned to the shader.

If the shader invocation is inactive, for example if the pixel has been discarded earlier in its execution, or if a pixel/sample invocation only exists to serve as a helper to a real pixel/sample for derivatives, this instruction does not alter the *dest* memory at all (silently).

Out of bounds addressing on u# causes nothing to be written to memory, except if the u# is structured, and byte offset into the struct (second component of the address) is causing the out of bounds access, then the entire contents of the UAV become undefined.

Out of bounds addressing on g# (the bounds of that particular g#, as opposed to all shared memory) causes the entire contents of all shared memory to become undefined.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

Because UAVs are available at all shader stages for Direct3D 11.1, this instruction applies to all shader stages for the Direct3D 11.1 runtime, which is available starting with Windows 8.

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

atomic_imin (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Atomic signed integer minimum to memory.

atomic_imin dest, dstAddress[.swizzle], src0[.select_component]

Item	Description
<i>dest</i>	[in] The components to compare to <i>src0</i> . This value must be an unordered access view (UAV) (u#). In the compute shader it can also be thread group shared memory (g#).
<i>dstAddress</i>	[in] The memory address.
<i>src0</i>	[in] The components to compare to <i>dest</i> .

Remarks

This instruction performs a single component 32-bit signed minimum of operand *src0* into *dest* at 32-bit per component address *dstAddress*, performed atomically.

The number of components taken from the address is determined by the dimensionality of *dest* u# or g#.

If *dest* is a u#, it can be declared as raw, typed or structured. If typed, it must be declared as UINT/SINT with the bound resource format being R32_UINT/_SINT.

If *dest* is g#, it must be declared as raw or structured.

Nothing is returned to the shader.

If the shader invocation is inactive, for example if the pixel has been discarded earlier in its execution, or if a pixel/sample invocation only exists to serve as a helper to a real pixel/sample for derivatives, this instruction does not alter the *dest* memory at all (silently).

Out of bounds addressing on u# causes nothing to be written to memory, except if the u# is structured, and byte offset into the struct (second component of the address) is causing the out of bounds access, then the entire contents of the UAV become undefined.

Out of bounds addressing on g# (the bounds of that particular g#, as opposed to all shared memory) causes the entire contents of all shared memory to become undefined.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

Because UAVs are available at all shader stages for Direct3D 11.1, this instruction applies to all shader stages for the Direct3D 11.1 runtime, which is available starting with Windows 8.

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

atomic_or (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Atomic bitwise OR to memory.

atomic_or dest, dstAddress[.swizzle], src0[.select_component]

Item	Description
<i>dest</i>	[in] The components to OR with <i>src0</i> . This value must be an unordered access view (UAV) (u#). In the compute shader it can also be thread group shared memory (g#).
<i>dstAddress</i>	[in] The memory address.
<i>src0</i>	[in] The components to OR with <i>dest</i> .

Remarks

This instruction performs a single component 32-bit bitwise OR of operand *src0* into *dest* at 32-bit per component address *dstAddress*, performed atomically.

The number of components taken from the address is determined by the dimensionality of *dest* u# or g#.

If *dest* is a u#, it can be declared as raw, typed or structured. If typed, it must be declared as UINT/SINT with the bound resource format being R32_UINT/_SINT.

If *dest* is g#, it must be declared as raw or structured.

Nothing is returned to the shader.

If the shader invocation is inactive, for example if the pixel has been discarded earlier in its execution, or if a pixel/sample invocation only exists to serve as a helper to a real pixel/sample for derivatives, this instruction does not alter the *dest* memory at all (silently).

Out of bounds addressing on u# causes nothing to be written to memory, except if the u# is structured, and the byte offset into the struct (second component of the address) is causing the out of bounds access, then the entire contents of the UAV become undefined.

Out of bounds addressing on g# (the bounds of that particular g#, as opposed to all shared memory) causes the entire contents of all shared memory to become undefined.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

Because UAVs are available at all shader stages for Direct3D 11.1, this instruction applies to all shader stages for the Direct3D 11.1 runtime, which is available starting with Windows 8.

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

atomic_umax (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Atomic unsigned integer maximum to memory.

atomic_umax dest, dstAddress[.swizzle], src0[.select_component]

Item	Description
<i>dest</i>	[in] The components to compare to <i>src0</i> . This value must be an unordered access view (UAV) (u#). In the compute shader it can also be thread group shared memory (g#).
<i>dstAddress</i>	[in] The memory address.
<i>src0</i>	[in] The components to compare to <i>dest</i> .

Remarks

This instruction performs a single component 32-bit unsigned maximum of operand *src0* into *dest* at 32-bit per component address *dstAddress*, performed atomically.

The number of components taken from the address is determined by the dimensionality of *dest* u# or g#.

If *dest* is a u#, it can be declared as raw, typed or structured. If typed, it must be declared as UINT/SINT with the bound resource format being R32_UINT/_SINT.

If *dest* is g#, it must be declared as raw or structured.

Nothing is returned to the shader.

If the shader invocation is inactive, for example if the pixel has been discarded earlier in its execution, or if a pixel/sample invocation only exists to serve as a helper to a real pixel/sample for derivatives, this instruction does not alter the *dest* memory at all (silently).

Out of bounds addressing on u# causes nothing to be written to memory, except if the u# is structured, and byte offset into the struct (second component of the address) is causing the out of bounds access, then the entire contents of the UAV become undefined.

Out of bounds addressing on g# (the bounds of that particular g#, as opposed to all shared memory) causes the entire contents of all shared memory to become undefined.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

Because UAVs are available at all shader stages for Direct3D 11.1, this instruction applies to all shader stages for the Direct3D 11.1 runtime, which is available starting with Windows 8.

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

atomic_umin (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Atomic unsigned integer minimum to memory.

atomic_umin dest, dstAddress[.swizzle], src0[.select_component]

Item	Description
<i>dest</i>	[in] The components to compare to <i>src0</i> . This value must be an unordered access view (UAV) (u#). In the compute shader it can also be thread group shared memory (g#).
<i>dstAddress</i>	[in] The memory address.
<i>src0</i>	[in] The components to compare to <i>dest</i> .

Remarks

This instruction performs a single component 32-bit unsigned minimum of operand *src0* into *dest* at 32-bit per component address *dstAddress*, performed atomically.

The number of components taken from the address is determined by the dimensionality of *dest* u# or g#.

If *dest* is a u#, it can be declared as raw, typed or structured. If typed, it must be declared as UINT/SINT with the bound resource format being R32_UINT/_SINT.

If *dest* is g#, it must be declared as raw or structured.

Nothing is returned to the shader.

If the shader invocation is inactive, for example if the pixel has been discarded earlier in its execution, or if a pixel/sample invocation only exists to serve as a helper to a real pixel/sample for derivatives, this instruction does not alter the *dest* memory at all (silently).

Out of bounds addressing on u# causes nothing to be written to memory, except if the u# is structured, and byte offset into the struct (second component of the address) is causing the out of bounds access, then the entire contents of the UAV become undefined.

Out of bounds addressing on g# (the bounds of that particular g#, as opposed to all shared memory) causes the entire contents of all shared memory to become undefined.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

Because UAVs are available at all shader stages for Direct3D 11.1, this instruction applies to all shader stages for the Direct3D 11.1 runtime, which is available starting with Windows 8.

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

atomic_xor (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Atomic bitwise XOR to memory.

atomic_xor dest, dstAddress[.swizzle], src0[.select_component]

Item	Description
<i>dest</i>	[in] The components to XOR with <i>src0</i> . This value must be an unordered access view (UAV) (u#). In the compute shader it can also be thread group shared memory (g#).
<i>dstAddress</i>	[in] The memory address.
<i>src0</i>	[in] The components to XOR with <i>dest</i> .

Remarks

This instruction performs a single component 32-bit bitwise XOR of operand *src0* into *dest* at 32-bit per component address *dstAddress*, performed atomically.

The number of components taken from the address is determined by the dimensionality of *dest* u# or g#.

If *dest* is a u#, it can be declared as raw, typed or structured. If typed, it must be declared as UINT/SINT with the bound resource format being R32_UINT/_SINT.

If *dest* is g#, it must be declared as raw or structured.

Nothing is returned to the shader.

If the shader invocation is inactive, for example if the pixel has been discarded earlier in its execution, or if a pixel/sample invocation only exists to serve as a helper to a real pixel/sample for derivatives, this instruction does not alter the *dest* memory at all (silently).

Out of bounds addressing on u# causes nothing to be written to memory, except if the u# is structured, and byte offset into the struct (second component of the address) is causing the out of bounds access, then the entire contents of the UAV become undefined.

Out of bounds addressing on g# (the bounds of that particular g#, as opposed to all shared memory) causes the entire contents of all shared memory to become undefined.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

Because UAVs are available at all shader stages for Direct3D 11.1, this instruction applies to all shader stages for the Direct3D 11.1 runtime, which is available starting with Windows 8.

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

bfi (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Given a bit range from the LSB of a number, place that number of bits in another number at any offset.

bfi dest[.mask], src0[.swizzle], src1[.swizzle], src2[.swizzle], src3[.swizzle]

Item	Description
<i>dest</i>	[in] The address of the results.
<i>src0</i>	[in] The bitfield width to take from <i>src2</i> .
<i>src1</i>	[in] The bitfield offset for replacing bits in <i>src3</i> .
<i>src2</i>	[in] The number the bits are taken from.
<i>src3</i>	[in] The number with bits to be replaced.

Remarks

The LSB 5 bits of *src0* provide the bitfield width (0-31) to take from *src2*.

The LSB 5 bits of *src1* provide the bitfield offset (0-31) to start replacing bits in the number read from *src3*.

syntax

Given width, offset:

```
bitmask = (((1 << width)-1) << offset) & 0xffffffff  
dest = ((src2 << offset) & bitmask) | (src3 & ~bitmask)
```

This instruction is used for packing integers or flags.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

bfrev (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Reverse a 32-bit number.

bfrev dest[.mask], src0[.swizzle]

Item	Description
<i>dest</i>	[in] The address for <i>src0</i> with bits reversed.
<i>src0</i>	[in] The 32-bit number to reverse.

Remarks

For example, given 0x12345678 the result would be 0x1e6a2c48.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

Shader Model 5 Assembly (DirectX HLSL)

bufinfo (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Query the element count on a buffer (but not the constant buffer).

bufinfo dest[.mask], srcResource

Item	Description
<i>dest</i>	[in] The address of the results.
<i>srcResource</i>	[in] Buffer, other than a constant Buffer, in an SRV (t#) or UAV (u#).

Remarks

All components in *dest* receive the integer number of elements in the buffer's shader resource view. The number of elements depends on the view parameters such as memory format.

For a typed buffer SRV or UAV, the return value is the number of elements in the view (where an element is one unit of the typed format).

For a raw buffer SRV or UAV, the return value is the number of bytes in the view.

For a structured buffer SRV or UAV, the return value is the number of structures in the view.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no

Shader Model	Supported
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

countbits (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Counts the number of bits set in a number.

countbits dest[.mask], src0[.swizzle]

Item	Description
<i>dest</i>	[in] The address of the results.
<i>src0</i>	[in] The input 32-bit number.

Remarks

This instruction can be used to compute shader input coverage percentage.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

Shader Model 5 Assembly (DirectX HLSL)

cut_stream (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

The geometry shader instruction that completes the current primitive topology at the specified stream, if any vertices have been emitted to it, and starts a new topology of the type declared by the geometry shader at that stream.

cut_stream streamIndex

Item	Description
<i>streamIndex</i>	[in] The stream index.

Remarks

When this instruction is executed, any previously emitted topology by the geometry shader invocation is completed. If there are not enough vertices emitted for the previous primitive topology, then they are discarded. Because the only available output topologies for the geometry shader are pointlist, linestrip and trianglestrip, there are never any leftover vertices.

streamIndex must be an immediate value [0..3] for a declared stream.

After the previous topology, if any, is completed, this instruction causes a new topology to begin, using the topology declared as the output for the geometry shader.

Restrictions

- This instruction applies to the geometry shader only.
- **cut_stream** can appear any number of times in the geometry shader, including within flow control.
- If the geometry shader ends and vertices have been emitted, the topology they are building is completed, as if a **cut_stream** instruction was executed as the last instruction.
- If streams have not been declared, you must use [cut](#) instead of **cut_stream**.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
			X		

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

dadd (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Component-wise double-precision add.

dadd[_sat] dest[.mask], [-]src0[_abs][.swizzle], [-]src1[_abs][.swizzle]

Item	Description
<i>dest</i>	[in] The address of the result of the operation.
<i>src0</i>	[in] The components to add with <i>src1</i> .
<i>src1</i>	[in] The components to add with <i>src0</i>

Remarks

The valid swizzles for the source parameters are `.xyzw`, `.xyxy`, `.zwxy`, `.zwzw`. The valid *dest* masks are `.xy`, `.zw`, and `.xyzw`. The following mappings are post-swizzle:

- *dest* is a double vec2 across (x 32LSB, y 32MSB) and (z 32LSB, w 32MSB).
 - *src0* is a double vec2 across (x 32LSB, y 32MSB) and (z 32LSB, w 32MSB).
 - *src1* is a double vec2 across (x 32LSB, y 32MSB) and (z 32LSB, w 32MSB).

The following table shows the results obtained when executing the instruction with various classes of numbers, assuming that neither overflow or underflow occurs.

F means finite-real number.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

dcl_function_body (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Declare a function body.

dcl_function_body fb#

Item	Description
fb#	[in] The label of the place where the function will appear.

Remarks

This instruction declares a unique function body whose code will appear later in the program at label fb#.

Function bodies are used in function table declarations. For more info, see [dcl_function_table](#).

In the hull shader and domain shader, where there are multiple phases (control point phase, fork phase, and join phase), all function bodies (label fb#) appear after all the phases, rather than being grouped by phase.

There is no limit to how many function bodies can be present.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no

Shader Model	Supported
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

dcl_function_table (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Declare a function table.

```
dcl_function_table ft# = {fb#, fb#, ...}
```

Item	Description
<i>ft</i>	[in] The function table entries.

Remarks

This function declares a function table as a set of function bodies that have been declared earlier.

This is like a C++ vtable except there is an entry per call site for an interface instead of per method.

There is no limit to how many function bodies can be listed in a function table.

It is valid for a given function body *fb#* to be referenced multiple times in one or more function tables, as a way of sharing common code.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no

Shader Model	Supported
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

dcl_hs_fork_phase_instance_count (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Declare the fork phase instance count in a hull shader fork phase.

dcl_hs_fork_phase_instance_count N

Item	Description
N	[in] The instance count (UINT greater than 0).

Remarks

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
	X				

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

dcl_hs_join_phase_instance_count (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Declare the join phase instance count in a hull shader join phase.

dcl_hs_join_phase_instance_count N

Item	Description
N	[in] The instance count (UINT greater than 0).

Remarks

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
	X				

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

dcl_hs_max_tessfactor (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Declare the maxTessFactor for the patch.

dcl_hs_max_tessfactor N

Item	Description
N	[in] The maxTessFactor.

Remarks

The maxTessFactor is a float32 value in the range {1.0 ... 64.0}.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
	X				

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

dcl_input vForkInstanceID (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Declare the instance ID in a hull shader fork phase.

dcl_input vForkInstanceID

Item	Description
<i>vForkInstanceID</i>	[in] The instance ID.

Remarks

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
	X				

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

dcl_input vGSInstanceId (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Enable geometry shader instancing.

dcl_input vGSInstanceId, instanceCount

Item	Description
<i>vGSInstanceId</i>	[in] The instance ID.
<i>instanceCount</i>	[in] The instance count.

Remarks

The *instanceCount* parameter of the declaration specifies how many instances the geometry shader should execute for each input primitive. The maximum value for *instanceCount* is 32.

The maximum number of vertices declared for output, via [dcl_maxOutputVertexCount](#), applies individually to each instance.

The instance count in this declaration, multiplied by the maximum vertex count per instance via [dcl_maxOutputVertexCount](#), must be <= 1024.

The amount of data that a given geometry shader instance can emit is 1024 scalars maximum, validated by counting up all scalars declared for input and multiplying by the declared output vertex count.

Use of geometry shader instancing effectively increases the total amount of data that can be emitted per input primitive. 1024 scalars for a single instance yields up to 1024*32 scalars of output data across all geometry shader instances for a single input primitive. However the more instances, the fewer vertices each instance can emit. A single instance (no instancing) can emit 1024 vertices. If you declare *32 instances it means each instance can only output $1024/32 = 32$ vertices.

The geometry shader instancing declaration makes available to the program a standalone 32-bit integer input register, *vGSInstanceId*. Each geometry shader instance is identified by the value contained in *vGSInstanceId* [0,1,2...].

vGSInstanceId is not part of the geometry shader input vertex array (e.g. 3 vertices when inputting a triangle). The *vGSInstanceId* register stands on its own, like *vPrimitiveID*.

When each geometry shader instance ends, there is an implicit cut in the output topology, so consecutive instances do not depend on each other.

Although hardware may execute each geometry shader instance in parallel, the output of all instances at the end is serialized as if all the instanced geometry shader invocations ran sequentially in a loop iterating `vGSInstanceID` from 0 to `instanceCount`-1, with implicit output topology cuts at the end of each instance.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
			X		

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

dcl_input vJoinInstanceID (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Declare the instance ID in a hull shader join phase.

dcl_input vJoinInstanceID

Item	Description
<code>vJoinInstanceID</code>	[in] The instance ID.

Remarks

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
	X				

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

dcl_input vOutputControlPointID (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Declare the output control point ID in a hull shader control point phase.

```
dcl_input vOutputControlPointID
```

Item	Description
<i>vOutputControlPointID</i>	[in] The output control point ID.

Remarks

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
	X				

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

dcl_input vThread (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Declare compute shader input IDs.

dcl_input vThread

Item	Description
<i>vThread</i>	[in] The 3-component unsigned 32-bit integer ID value. One of: <ul style="list-style-type: none">• vThreadId.xyz• vThreadGroupID.xyz• vThreadIdInGroup.xyz• vThreadIdInGroupFlattened

dcl_input is an existing declaration in other shader stages. It is used in the compute shader to declare the various 3-component unsigned 32-bit integer ID values unique to the compute shader. They are:

- vThreadId.xyz
- vGroupID.xyz
- vThreadIdInGroup.xyz
- vThreadIdInGroupFlattened (single component)

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
					X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no

Shader Model	Supported
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

dcl_input_control_point_count (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Declare the hull shader input control point count in the hull shader declaration section.

```
dcl_input_control_point_count N
```

Item	Description
N	[in] The input control point count {1..32}.

Remarks

At least 1 input control point is required; it can be empty if it is not needed.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
	X				

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

Shader Model 5 Assembly (DirectX HLSL)

dcl_interface (sm5 - asm)

Article • 03/09/2021 • 2 minutes to read

Declare function table pointers (interfaces).

```
dcl_interface fp#[arraySize][numCallSites] = {ft#, ft#, ...}
```

Item	Description
<i>fp#</i>	[in] The function table pointers.

Remarks

Each interface needs to be bound from the API before the shader is usable. Binding gives a reference to one of the function tables so that the method slots can be filled in. The compiler will not generate pointers for unreferenced objects.

A function table pointer has a full set of method slots to avoid the extra level of indirection that a C++ pointer-to- pointer-to-vtable representation would require. That would also require that this pointers be 5-tuples. In the HLSL virtual inlining model it's always known what global variable/input is used for a call so we can set up tables per root object.

Function pointer declarations indicate which function tables are legal to use with them. This also allows derivation of method correlation information.

The first [] of an interface declaration is the array size. If dynamic indexing is used the declaration will indicate that as shown. An array of interface pointers can be indexed statically also, it is not required that arrays of interface pointers mean dynamic indexing.

Numbering of interface pointers starts at 0 for the first declaration and subsequently takes array size into account, so the first pointer after a four entry array `fp0[4][1]` would be `fp4[][]`.

The second [] of an interface declaration is the number of call sites, which must match the number of bodies in each table referenced in the declaration.

There are no bounds to how many function table (ft#) choices can be listed in an interface declaration.

A given function table (ft#) can appear more than once in one or more interface declarations.

Restrictions

- The number of object sites in a shader, which is the sum across all *fp#* declarations of their [arraySize] declarations, must be no more than 253. This number corresponds to how many **this** pointers can be present. The runtime enforces this 253 limit to keep a bound on the size of the DDI for communicating this pointer data.
- The number of call sites in a shader, which is the sum across all fcall statements of the number of potential branch targets, must be no more than 4096.

For example, an **fcall** that uses a static index for the first *fp[]/[]* dimension counts as one:

```
fcall fp0[0][0] // +1
```

An **fcall** that uses a dynamic index counts as the number of elements in the array (first [] of **dcl_interface**):

```
dcl_interface_dynamicindexed fp1[2][1] = {ft2, ft3, ft4} ...
```

```
fcall fp1[r0.z + 0][1] // +2
```

This limit helps some implementations easily fit tables of function body selections in constant buffer-like storage.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no

Shader Model	Supported
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

cs_4_0 and cs_4_1 support this instruction for UAV and SRV.

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

dcl_interface_dynamicindexed (sm5 - asm)

Article • 03/09/2021 • 2 minutes to read

Declare function table pointers (interfaces).

```
dcl_interface_dynamicindexed fp#[arraySize][numCallSites] = {ft#, ft#, ...}
```

Item	Description
<i>fp#</i>	[in] The function table pointers.

Remarks

Each interface needs to be bound from the API before the shader is usable. Binding gives a reference to one of the function tables so that the method slots can be filled in. The compiler will not generate pointers for unreferenced objects.

A function table pointer has a full set of method slots to avoid the extra level of indirection that a C++ pointer-to- pointer-to-vtable representation would require. That would also require that these pointers be 5-tuples. In the HLSL virtual inlining model it's always known what global variable/input is used for a call so we can set up tables per root object.

Function pointer declarations indicate which function tables are legal to use with them. This also allows derivation of method correlation information.

The first [] of an interface declaration is the array size. If dynamic indexing is used the declaration will indicate that as shown. An array of interface pointers can be indexed statically also, it is not required that arrays of interface pointers mean dynamic indexing.

Numbering of interface pointers starts at 0 for the first declaration and subsequently takes array size into account, so the first pointer after a four entry array *fp0[4][1]* would be *fp4[][]*.

The second [] of an interface declaration is the number of call sites, which must match the number of bodies in each table referenced in the declaration.

There are no bounds to how many function table (ft#) choices can be listed in an interface declaration.

A given function table (ft#) can appear more than once in one or more interface declarations.

Restrictions

- The number of object sites in a shader, which is the sum across all *fp#* declarations of their [arraySize] declarations, must be no more than 253. This number corresponds to how many **this** pointers can be present. The runtime enforces this 253 limit to keep a bound on the size of the DDI for communicating this pointer data.
- The number of call sites in a shader, which is the sum across all fcall statements of the number of potential branch targets, must be no more than 4096.

For example, an **fcall** that uses a static index for the first *fp[]/[]* dimension counts as one:

```
fcall fp0[0][0] // +1
```

An **fcall** that uses a dynamic index counts as the number of elements in the array (first [] of **dcl_interface**):

```
dcl_interface_dynamicindexed fp1[2][1] = {ft2, ft3, ft4} ...
```

```
fcall fp1[r0.z + 0][1] // +2
```

This limit helps some implementations easily fit tables of function body selections in constant buffer-like storage.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no

Shader Model	Supported
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

cs_4_0 and cs_4_1 support this instruction for UAV and SRV.

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

dcl_output oMask (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Declare an output register to be written by the shader.

```
dcl_output o#[.mask]
```

Item	Description
<i>o#</i>	[in] The output register. <ul style="list-style-type: none"># is an name that identifies the register.<i>[.mask]</i> is an optional component mask (.xyzw) that specifies which of the register components to use.

Remarks

Example:

```
dcl_output oMask[3].xyz
```

Restrictions

- The component mask can be any subset of [xyzw]. However, leaving gaps between components wastes space.
- It is legal to declare a superset of the component mask declared for input by the next stage. However mutually exclusive masks are not allowed. The vertex shader outputting o3.xy, means the pixel shader inputting v3.z is invalid, but inputting v3.x or v3.y or v3.xy is valid.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

dcl_output_control_point_count (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Declares the hull shader output control point count.

dcl_output_control_point_count N

Item	Description
N	[in] An integer value from 0 to 32 that specifies the output control point count.

Remarks

The hull shader can output 0 control points if they are not needed.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
Declarations Section					

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

Shader Model 5 Assembly (DirectX HLSL)

dcl_resource_raw (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Declare a shader resource input and assign it to a t# - a placeholder register for the resource.

```
dcl_resource_raw dstSRV
```

Item	Description
<i>dstSRV</i>	[in] A t# register declared as a reference to a ShaderResourceView of a raw buffer.

Remarks

The contents of the structure have no type; operations performed on the memory may implicitly interpret the data as having a type.

Instructions that reference a raw t# take a 1D address, an unsigned 32-bit value specifying the byte offset to a 32-bit aligned location in the Buffer. The address must be a multiple of 4 (bytes).

Views bound to t# declared as raw must have RAW specified on their creation; otherwise behavior when accessed from a shader is undefined.

cs_4_0 and cs_4_1 support this instruction.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no

Shader Model	Supported
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

dcl_resource_structured (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Declare a shader resource input and assign it to a t# - a placeholder register for the resource.

dcl_resource_structured dstSRV, structByteStride

Item	Description
<i>dstSRV</i>	[in] A t# register declared as a reference to a ShaderResourceView of a structured buffer with the specified stride that must be bound to SRV slot # at the API.
<i>structByteStride</i>	[in] A uint that specifies the size of the structure in bytes in the buffer being declared. This value must be greater than zero.

Remarks

The contents of the structure have no type; operations performed on the memory may implicitly interpret the data as having a type.

Instructions that reference a structured t# take a 2D address, where the first component picks [struct], and the second component picks [offset within struct, multiple of 32-bits].

cs_4_0 and cs_4_1 support this instruction.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no

Shader Model	Supported
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

dcl_stream (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Declare a geometry shader output stream.

dcl_stream mN

Item	Description
<i>mN</i>	[in] Stream where <i>N</i> is 0..3 (m0..m3).

Remarks

A given stream can only be declared once.

If no streams are declared, output and output topology declarations are assumed to be for stream 0.

The first **dcl_stream** cannot appear after any **dcl_output** or **dcl_outputTopology** statements.

Any **dcl_output** or **dcl_outputTopology** statements after any **dcl_stream m#** statement define the outputs for stream *m#*.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
			X		

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no

Shader Model	Supported
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

dcl_tessellator_domain (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Declare the tessellator domain in a hull shader declaration section, and the domain shader.

dcl_tessellator_domain domain

Item	Description
<i>domain</i>	[in] The domain. One of: <ul style="list-style-type: none">• domain_isoline• domain_tri• domain_quad

Remarks

Behavior is undefined if the hull shader and domain shader provide mismatching domains or any other conflicting declarations.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
		X			

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no

Shader Model	Supported
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

dcl_tessellator_output_primitive (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Declare the tessellator output primitive type in a hull shader declaration section.

dcl_tessellator_output_primitive type

Item	Description
<i>type</i>	[in] The output primitive type. One of: <ul style="list-style-type: none">• output_point• output_line• output_triangle_cw• output_triangle_ccw

Remarks

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
Declarations Section					

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no

Shader Model	Supported
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

dcl_tessellator_partitioning (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Declare the tessellator partitioning in a hull shader declaration section.

dcl_tessellator_partitioning mode

Item	Description
<i>mode</i>	[in] The tessellator partitioning mode. One of: <ul style="list-style-type: none">partitioning_integerpartitioning_pow2partitioning_fractional_oddpartitioning_fractional_even

Remarks

From the hardware point of view, _pow2 behaves just like _integer. It is up to the HLSL shader author and/or compiler code to round TessFactors to powers of 2.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
Declarations Section					

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no

Shader Model	Supported
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

dcl_tgsm_raw (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Declare a reference to a region of shared memory space available to the compute shader's thread group.

dcl_tgsm_raw g#, byteCount

Item	Description
<i>g#</i>	[in] A reference to a block of size <i>byteCount</i> of untyped shared memory.
<i>byteCount</i>	[in] Must be a multiple of 4.

Remarks

The total storage for all *g#* must be \leq the amount of shared memory available per thread group, which is 32kB.

In an extreme case, you can declare 8192 total *g#*'s, each with a *byteCount* of 4.

In the opposite extreme, you can declare a single *g#* with a *byteCount* of 32768.

ⓘ Note

cs_4_0 and cs_4_1 supports `dcl_tgsm_structured`, but not `dcl_tgsm_raw`.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes

Shader Model	Supported
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

dcl_tgsm_structured (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Declare a reference to a region of shared memory space available to the compute shader's thread group. The memory is viewed as an array of structures.

dcl_tgsm_structured g#, structByteStride, structCount

Item	Description
<i>g#</i>	[in] A reference to a block of shared memory of size <i>structByteStride</i> * <i>structCount</i> bytes.
<i>structByteStride</i>	[in] The structure stride. This value is a uint in bytes and must be a multiple of 4.
<i>structCount</i>	[in] The number of structures.

Remarks

The total storage for all *g#* must be <= the amount of shared memory available per thread group, which is 32kB, or 8192 32-bit scalars.

In an extreme case, you can declare 8192 total *g#*'s, if each has a *structByteStride* of 4 and a *structCount* of 1.

In the opposite extreme, you can declare a single *g#* with a structure stride of 32kB and a structure count of 1.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
					X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no

Shader Model	Supported
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

dcl_thread_group (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Declare thread group size.

```
dcl_thread_group x, y, z
```

Item	Description
x	[in] An unsigned 32-bit integer. $1 \leq x \leq 1024$
y	[in] An unsigned 32-bit integer. $1 \leq y \leq 1024$
z	[in] An unsigned 32-bit integer. $1 \leq z \leq 64$

Remarks

$x * y * z \leq 1024$

This thread group declaration must appear once in a compute shader.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
					X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

dcl_uav_raw (sm5 - asm)

Article • 03/09/2021 • 2 minutes to read

Declare an unordered access view (UAV) for use by a shader.

```
dcl_uav_raw[_glc] dstUAV
```

Item	Description
<i>dstUAV</i>	[in] The UAV.

Remarks

dstUAV is a u# register declared as a reference to an UnorderedAccessView of a Buffer, where the buffer appears as a simple 1D array of 32-bit untyped entries.

Operations performed on the memory may implicitly interpret the data as having a type.

The _glc flag means "globally coherent". The absence of _glc means the UAV is being declared only as "group coherent" in the compute shader, or "locally coherent" in a single pixel shader invocation.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

Because UAVs are available at all shader stages for Direct3D 11.1, this instruction applies to all shader stages for the Direct3D 11.1 runtime, which is available starting with Windows 8.

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

ⓘ Note

This instruction is supported in cs_4_0 and cs_4_1.

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

dcl_uav_structured (sm5 - asm)

Article • 03/09/2021 • 2 minutes to read

Declare an unordered access view (UAV) for use by a shader.

dcl_uav_structured[_glc] dstUAV, structByteStride

Item	Description
<i>dstUAV</i>	[in] The UAV.
<i>structByteStride</i>	[in] The size of the structure in bytes.

Remarks

dstUAV is a u# register declared as a reference to an UnorderedAccessView of a structured buffer with the specified stride that must be bound to UAV slot # at the API.

The contents of the structure have no type; operations performed on the memory may implicitly interpret the data as having a type.

structByteStride is the size of the structure in bytes in the buffer being declared. This value must be greater than zero. *structByteStride* is of type uint, and must be a multiple of 4.

Instructions that reference a structured u# take a 2D address, where the first component picks [struct], and the second component picks [offset within struct, in aligned bytes].

The _glc flag means for "globally coherent". The absence of _glc means the UAV is being declared only as "group coherent" in the compute shader, or "locally coherent" in a single pixel shader invocation.

The _opc flag is the order preserving counter. It indicates that if a UAV is bound to slot # (u#), it must have been created with the COUNTER flag. This means that [imm_atomic_alloc](#) or [imm_atomic_consume](#) operations in the shader manipulate a counter whose values can be used in the shader as a permanent reference to a location in the UAV. Data cannot be reordered after the shader is over.

The absence of the _opc flag means that if the shader uses [imm_atomic_alloc](#) or [imm_atomic_consume](#) instructions and a UAV is bound to slot # (u), it must have been created with the APPEND flag, which provides a counter that does not guarantee order is preserved after the shader invocation.

If the `_opc` flag is absent and the shader does not contain `imm_atomic_alloc` or `imm_atomic_consume` instructions, a UAV bound to slot # (u) is permitted to have been created with the COUNTER flag (the counter will go unused by this shader), no flag (no counter), but not with the APPEND flag.

ⓘ Note

`cs_4_0` and `cs_4_1` supports `dcl_tgsm_structured`, but not `dcl_tgsm_raw`.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

Because UAVs are available at all shader stages for Direct3D 11.1, this instruction applies to all shader stages for the Direct3D 11.1 runtime, which is available starting with Windows 8.

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

ⓘ Note

This instruction is supported in `cs_4_0` and `cs_4_1`.

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

dcl_uav_typed (sm5 - asm)

Article • 03/09/2021 • 2 minutes to read

Declare an unordered access view (UAV) for use by a shader.

```
dcl_uav_typed[_glc] dstUAV, dimension, type
```

Item	Description
<i>dstUAV</i>	[in] The UAV.
<i>dimension</i>	[in] Specifies how many dimensions the instructions accessing the UAV are providing.
<i>type</i>	[in] The type of the UAV.

Remarks

dstUAV is a u# register being declared as a reference to an UnorderedAccessView that must be bound to UAV slot # at the API.

The dimension must be buffer, Texture1D, Texture1DArray, Texture2D, Texture2DArray, or Texture3D. This indicates how many dimensions any instructions accessing the UAV are providing: 1 (Texture1D, Buffer), 2 (Texture1DArray, Texture2D) or 3 (Texture2DArray, Texture3D).

Type is {UNORM,SNORM,UINT,SINT,FLOAT}. Operations done with the declared u# must be compatible with the type declared here, and the UAV bound to slot # must also have the same type.

The _glc flag stands for "globally coherent". The absence of _glc means the UAV is being declared only as "group coherent" in the compute shader, or "locally coherent" in a single pixel shader invocation.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

Because UAVs are available at all shader stages for Direct3D 11.1, this instruction applies to all shader stages for the Direct3D 11.1 runtime, which is available starting with Windows 8.

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

ⓘ Note

This instruction is not supported in compute shader 4.x.

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

ddiv (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Computes a component-wise double-precision division.

```
ddiv[_sat] dest[.mask], [-]src0[.abs][.swizzle], [-]src1[.abs][.swizzle]
```

Item	Description
<i>dest</i>	[in] The result of the operation. The result value must be accurate to 0.5 ULP.
<i>src0</i>	[in] The dividend.
<i>src1</i>	[in] The divisor.

Remarks

The DDIV instruction will be emitted by the HLSL compiler whenever the division operator is used with doubles. The accuracy of this instruction will be required to be 0.5 ULP.

Shaders that use this instruction will be marked with a shader flag that will cause them to fail to bind unless all the following conditions are met.

- The system supports DirectX 11.1.
- The system includes a WDDM 1.2 driver.
- The driver reports support for this instruction via `D3D11_FEATURE_DATA_D3D11_OPTIONS.ExtendedDoublesShaderInstructions` set to `TRUE`.

The following table shows the results obtained when executing the instruction with various classes of numbers, assuming that neither overflow or underflow occurs.

In this table F means finite-real number.

src0 src1 ->	-inf	-F	-1.0	-0	+0	+1.0	+F	+inf	NaN
-inf	NaN	+inf	+inf	+inf	-inf	-inf	-inf	NaN	NaN
-F	+0	+F	-src0	+inf	-inf	src0	-F	-0	NaN
-0	+0	+0	+0	NaN	NaN	-0	-0	-0	NaN
+0	-0	-0	-0	NaN	NaN	+0	+0	+0	NaN

<code>src0 src1 -></code>	<code>-inf</code>	<code>-F</code>	<code>-1.0</code>	<code>-0</code>	<code>+0</code>	<code>+1.0</code>	<code>+F</code>	<code>+inf</code>	<code>NaN</code>
<code>+F</code>	<code>-0</code>	<code>-F</code>	<code>-src0</code>	<code>-inf</code>	<code>+inf</code>	<code>src0</code>	<code>+F</code>	<code>+0</code>	<code>NaN</code>
<code>+inf</code>	<code>Nan</code>	<code>-inf</code>	<code>-inf</code>	<code>-inf</code>	<code>+inf</code>	<code>+inf</code>	<code>+inf</code>	<code>NaN</code>	<code>NaN</code>
<code>NaN</code>	<code>NaN</code>	<code>NaN</code>	<code>NaN</code>	<code>NaN</code>	<code>NaN</code>	<code>NaN</code>	<code>NaN</code>	<code>NaN</code>	<code>NaN</code>

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

deq (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Component-wise double-precision equality comparison.

```
deq[_sat] dest[.mask], [-]src0[_abs][.swizzle], [-]src1[_abs][.swizzle]
```

Item	Description
<i>dest</i>	[in] The address of the results of the operation.
<i>src0</i>	[in] The components to compare to <i>src1</i> .
<i>src1</i>	[in] The components to compare to <i>src0</i> .

Remarks

This instruction performs the double-precision floating-point comparison ($src0 == src1$) for each component and writes the result to *dest*.

If the comparison is true, then 32-bit 0xFFFFFFFF is returned for that component. Otherwise 32-bit 0x00000000 is returned.

Comparison with NaN returns false.

The valid *dest* masks are any one or 2 components. That is: .x, .y, .z, .w, .xy, .xz, .xw, .yz, .yw, .zw. The first *dest* component in the mask receives the 32-bit result for the first double comparison. The second component in the mask, if present, receives the 32-bit result for the second double comparison.

The valid swizzles for the source parameters are .xyzw, .xyxy, .zwxy, .zwzw. The following *src* mappings are post-swizzle:

- *src0* is a double vec2 across (x 32LSB, y 32MSB) and (z 32LSB, w 32MSB).
- *src1* is a double vec2 across (x 32LSB, y 32MSB) and (z 32LSB, w 32MSB).

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

deriv_rtx_coarse (sm5 - asm)

Article • 03/09/2021 • 2 minutes to read

Computes the rate of change of components.

```
deriv_rtx_coarse[_sat] dest[.mask], [-]src0[_abs][.swizzle],
```

Item	Description
<i>dest</i>	[in] The address of the results of the operation.
<i>src0</i>	[in] The components in the operation.

Remarks

This instruction computes the rate of change of contents of each float32 component of *src0* (post-swizzle), with regard to RenderTarget x direction (rtx) or RenderTarget y direction (see [deriv_rty_coarse](#)). Only a single x,y derivative pair is computed for each 2x2 stamp of pixels.

The data in the current pixel shader invocation may or may not participate in the calculation of the requested derivative, because the derivative will be calculated only once per 2x2 quad. For example, the x derivative could be a delta from the top row of pixels, and the y direction ([deriv_rty_coarse](#)) could be a delta from the left column of pixels. The exact calculation is up to the hardware vendor. There is also no specification dictating how the 2x2 quads will be aligned or tiled over a primitive.

Derivatives are calculated at a coarse level, once per 2x2 pixel quad. This instruction and [deriv_rty_coarse](#) are alternatives to [deriv_rtx_fine](#) and [deriv_rty_fine](#). These _coarse and _fine derivative instructions are a replacement for [deriv_rtxderiv_rty](#) from previous shader models .

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

deriv_rtx_fine (sm5 - asm)

Article • 03/09/2021 • 2 minutes to read

Computes the rate of change of components.

```
deriv_rtx_fine[_sat] dest[.mask], [-]src0[_abs][.swizzle],
```

Item	Description
<i>dest</i>	[in] The address of the results of the operation.
<i>src0</i>	[in] The components in the operation.

Remarks

This instruction computes the rate of change of contents of each float32 component of *src0* (post-swizzle), with regard to RenderTarget x direction (rtx) or RenderTarget y direction (see [deriv_rty_fine](#)). Each pixel in the 2x2 stamp gets a unique pair of x/y derivative calculations

The data in the current pixel shader invocation always participates in the calculation of the requested derivative. In the 2x2 pixel quad the current pixel falls within, the x derivative is the delta of the row of 2 pixels including the current pixel. The y derivative is the delta of the column of 2 pixels including the current pixel. There is no specification dictating how the 2x2 quads will be aligned or tiled over a primitive.

Derivatives are calculated at a fine level (unique calculation of the x/y derivative pair for each pixel in a 2x2 quad). This instruction and [deriv_rty_fine](#) are alternatives to [deriv_rtx_coarse](#) and [deriv_rty_coarse](#). These _coarse and _fine derivative instructions are a replacement for [deriv_rtx](#). These _coarse and _fine derivative instructions are a replacement for [deriv_rtx](#) and [deriv_rty](#) from previous shader models.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

deriv_rty_coarse (sm5 - asm)

Article • 03/09/2021 • 2 minutes to read

Computes the rate of change of components.

```
deriv_rty_coarse[_sat] dest[.mask], [-]src0[_abs][.swizzle],
```

Item	Description
<i>dest</i>	[in] The address of the results of the operation.
<i>src0</i>	[in] The components in the operation.

Remarks

For more information, see [deriv_rtx_coarse](#).

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

Shader Model 5 Assembly (DirectX HLSL)

deriv_rty_fine (sm5 - asm)

Article • 03/09/2021 • 2 minutes to read

Computes the rate of change of components.

```
deriv_rty_fine[_sat] dest[.mask], [-]src0[_abs][.swizzle],
```

Item	Description
<i>dest</i>	[in] The address of the results of the operation.
<i>src0</i>	[in] The components in the operation.

Remarks

For more information, see [deriv_rtx_fine](#).

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

Shader Model 5 Assembly (DirectX HLSL)

dfma (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Performs a fused-multiply add.

```
dfma[_sat] dest[.mask], [-]src0[_abs][.swizzle], [-]src1[_abs][.swizzle],[-]src2[_abs][.swizzle]
```

Item	Description
<i>dest</i>	[in] The address of the result of the operation. The result value must be accurate to 0.5 ULP. $dest = src0 * src1 + src2$
<i>src0</i>	[in] The components to multiply with <i>src1</i> .
<i>src1</i>	[in] The components to multiply with <i>src0</i> .
<i>src2</i>	[in] The components to add to $src0 * src1$.

Remarks

Shaders that use this instruction will be marked with a shader flag that will cause them to fail to bind unless all the following conditions are met.

- The system supports DirectX 11.1.
- The system includes a WDDM 1.2 driver.
- The driver reports support for this instruction via `D3D11_FEATURE_DATA_D3D11_OPTIONS.ExtendedDoublesShaderInstructions` set to `TRUE`.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
sm5	

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

dge (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Component-wise double-precision greater-than-or-equal comparison.

dge[_sat] dest[.mask], [-]src0[_abs][.swizzle], [-]src1[_abs][.swizzle]

Item	Description
<i>dest</i>	[in] The address of the results of the operation.
<i>src0</i>	The components to compare to <i>src1</i> .
<i>src1</i>	The components to compare to <i>src0</i> .

Remarks

This instruction performs the double-precision floating-point comparison ($src0 \geq src1$) for each component, and writes the result to *dest*.

If the comparison is true, then 32-bit 0xFFFFFFFF is returned for that component. Otherwise 32-bit 0x00000000 is returned.

Comparison with NaN returns false.

The valid *dest* masks are any one or two components. That is: .x, .y, .z, .w, .xy, .xz, .xw, .yz, .yw, .zw. The first *dest* component in the mask receives the 32-bit result for the first double comparison. The second component in the mask, if present, receives the 32-bit result for the second double comparison.

The valid swizzles for the source parameters are .xyzw, .xyxy, .zwxy, .zwzw. The following *src* mappings are post-swizzle:

- *src0* is a double vec2 across (x 32LSB, y 32MSB) and (z 32LSB, w 32MSB).
- *src1* is a double vec2 across (x 32LSB, y 32MSB) and (z 32LSB, w 32MSB).

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

dlt (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Component-wise double-precision less-than comparison.

dlt[_sat] dest[.mask], [-]src0[_abs][.swizzle], [-]src1[_abs][.swizzle]

Item	Description
<i>dest</i>	[in] The address of the results of the operation.
<i>src0</i>	[in] The components to compare to <i>src1</i> .
<i>src1</i>	[in] The components to compare to <i>src0</i> .

Remarks

This instruction performs the double-precision floating-point comparison ($src0 < src1$) for each component and writes the result to *dest*.

If the comparison is true, then 32-bit 0xFFFFFFFF is returned for that component. Otherwise 32-bit 0x00000000 is returned.

Comparison with NaN returns false.

The valid *dest* masks are any one or two components. That is: .x, .y, .z, .w, .xy, .xz, .xw, .yz, .yw, .zw. The first *dest* component in the mask receives the 32-bit result for the first double comparison. The second component in the mask (if present) receives the 32-bit result for the second double comparison.

The valid swizzles for the source parameters are .xyzw, .xyxy, .zwxy, .zwzw. The following *src* mappings are post-swizzle:

- *src0* is a double vec2 across (x 32LSB, y 32MSB) and (z 32LSB, w 32MSB).
- *src1* is a double vec2 across (x 32LSB, y 32MSB) and (z 32LSB, w 32MSB).

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

dmax (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Component-wise double-precision maximum.

```
dmax[_sat] dest[.mask], [-]src0[_abs][.swizzle], [-]src1[_abs][.swizzle]
```

Item	Description
<i>dest</i>	[in] The address of the results of the operation. $dest = src0 >= src1 ? src0 : src1$ \geq is used instead of $>$ so that if $\min(x,y) = x$ then $\max(x,y) = y$.
<i>src0</i>	[in] The value to compare with <i>src1</i> .
<i>src1</i>	[in] The value to compare with <i>src0</i> .

Remarks

NaN has special handling. If one source operand is NaN, then the other source operand is returned. The choice is made per-component. If both are NaN, any NaN representation is returned.

The valid swizzles for the source parameters are .xyzw, .xyxy, .zwxy, .zwzw. The valid *dest* masks are .xy, .zw, and .xyzw. The following *src* mappings are post-swizzle:

- *dest* is a double vec2 across (x 32LSB, y 32MSB) and (z 32LSB, w 32MSB).
- *src0* is a double vec2 across (x 32LSB, y 32MSB) and (z 32LSB, w 32MSB).
- *src1* is a double vec2 across (x 32LSB, y 32MSB) and (z 32LSB, w 32MSB).

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

dmin (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Component-wise double-precision minimum.

```
dmin[_sat] dest[.mask], [-]src0[_abs][.swizzle], [-]src1[_abs][.swizzle]
```

Item	Description
<i>dest</i>	[in] The address of the results of the operation. $dest = src0 < src1 ? src0 : src1$ < is used instead of \leq so that if $\min(x,y) = x$ then $\max(x,y) = y$.
<i>src0</i>	[in] The components to compare with <i>src1</i> .
<i>src1</i>	[in] The components to compare with <i>src0</i> .

Remarks

NaN has special handling. If one source operand is NaN, then the other source operand is returned. The choice is made per-component. If both are NaN, any NaN representation is returned.

The valid swizzles for the source parameters are .xyzw, .xyxy, .zwxy, .zwzw. The valid *dest* masks are .xy, .zw, and .xyzw. The following *src* mappings are post-swizzle:

- *dest* is a double vec2 across (x 32LSB, y 32MSB) and (z 32LSB, w 32MSB).
- *src0* is a double vec2 across (x 32LSB, y 32MSB) and (z 32LSB, w 32MSB).
- *src1* is a double vec2 across (x 32LSB, y 32MSB) and (z 32LSB, w 32MSB).

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

dmov (sm5 - asm)

Article • 03/09/2021 • 2 minutes to read

Component-wise move.

```
dmov[_sat] dest[.mask], [-]src0[_abs][.swizzle]
```

Item	Description
<i>dest</i>	[in] The move destination. <i>dest</i> = <i>src0</i> .
<i>src0</i>	[in] The components to be moved.

Remarks

The modifiers, other than swizzle, assume the data is floating point. The absence of modifiers moves data without altering bits.

The valid swizzles for the source parameters are .xyzw, .xyxy, .zwxy, .zwzw. The following *src* mappings are post-swizzle:

- *src0* is a double vec2 across (x 32LSB, y 32MSB) and (z 32LSB, w 32MSB).
- *src1* is a double vec2 across (x 32LSB, y 32MSB) and (z 32LSB, w 32MSB).

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no

Shader Model	Supported
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

dmovec (sm5 - asm)

Article • 03/09/2021 • 2 minutes to read

Component-wise conditional move.

dmovec[_sat] dest[.mask], src0[.swizzle], [-]src1[_abs][.swizzle], [-]src2[_abs][.swizzle],

Item	Description
<i>dest</i>	[in] The move destination. If <i>src0</i> , then <i>dest</i> = <i>src1</i> else <i>dest</i> = <i>src2</i> .
<i>src0</i>	[in] The components to test the condition against.
<i>src1</i>	[in] The components to move if the condition is true.
<i>src2</i>	[in] The components to move if the condition is false.

Remarks

The following example shows how to use this instruction.

syntax

```
if(the dest mask contains .xy)
{
    if(the first 32-bit component of src0, post-swizzle,
       has any bit set)
    {
        copy the first double from src1 (post swizzle)
        into dest.xy
    }
    else
    {
        copy the first double from src2 (post swizzle)
        into dest.xy
    }
}
if(the dest mask contains .zw)
{
    if(the second 32-bit component of src0, post-swizzle,
       has any bit set)
    {
        copy the second double from src1 (post swizzle)
        into dest.zw
    }
    else
    {
```

```

        copy the second double from src2 (post swizzle)
        into dest.zw
    }
}

```

The valid masks for *dest* are *.xy*, *.zw*, *.xyzw*.

The valid swizzles for *src0* are anything. The first two components post-swizzle are used to identify two 32-bit condition values.

The valid swizzles for *src1* and *src2* containing doubles are *.xyzw*, *.xyxy*, *.zwxy*, *.zwzw* are *.xy*, *.zw*, and *.xyzw*.

The following *src* mappings below are post-swizzle:

- *dest* is a double vec2 across (x 32LSB, y 32MSB) and (z 32LSB, w 32MSB).
- *src0* is a 32bit/component vec2 across x and y (zw ignored).
- *src1* is a double vec2 across (x 32LSB, y 32MSB) and (z 32LSB, w 32MSB).
- *src2* is a double vec2 across (x 32LSB, y 32MSB) and (z 32LSB, w 32MSB).

The modifiers on *src1* and *src2*, other than swizzle, assume the data is double. The absence of modifiers moves data without altering bits.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

dmul (sm5 - asm)

Article • 04/26/2021 • 2 minutes to read

Component-wise double-precision multiply.

dmul[_sat] dest[.mask], [-]src0[_abs][.swizzle], [-]src1[_abs][.swizzle]

Item	Description
<i>dest</i>	[in] The address of the result of the operation. $dest = src0 * src1$
<i>src0</i>	[in] The components to multiply with <i>src1</i> .
<i>src1</i>	[in] The components to multiply with <i>src0</i> .

Remarks

The valid swizzles for the source parameters are .xyzw, .xyxy, .zwxy, .zwzw. The valid *dest* masks are .xy, .zw, and .xyzw. The following *src* mappings are post-swizzle:

- *dest* is a double vec2 across (x 32LSB, y 32MSB) and (z 32LSB, w 32MSB).
- *src0* is a double vec2 across (x 32LSB, y 32MSB) and (z 32LSB, w 32MSB).
- *src1* is a double vec2 across (x 32LSB, y 32MSB) and (z 32LSB, w 32MSB).

The following table shows the results obtained when executing the instruction with various classes of numbers, assuming that neither overflow or underflow occurs.

F means finite-real number.

src0 src1->	-inf	-F	-1.0	-0	+0	+1.0	+F	+inf	NaN
-inf	+inf	+inf	+inf	NaN	NaN	-inf	-inf	-inf	NaN
-F	+inf	+F	-src0	+0	-0	src0	-F	-inf	NaN
-1.0F	+inf	-src1	+1.0	+0	-0	-1.0	-src1	-inf	NaN
-0	NaN	+0	+0	+0	-0	-0	-0	NaN	NaN
+0	NaN	-0	-0	-0	+0	+0	+0	NaN	NaN
+1.0	-inf	src1	-1.0	-0	+0	+1	src1	+inf	NaN
+F	-inf	-F	-src0	-0	+0	src0	+F	+inf	NaN

<code>src0</code>	<code>src1-></code>	-inf	-F	-1.0	-0	+0	+1.0	+F	+inf	NaN
+inf		-inf	-inf	-inf	NaN	NaN	+inf	+inf	+inf	NaN
NaN		NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

dne (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Component-wise double-precision not equality comparison.

dne[_sat] dest[.mask], [-]src0[_abs][.swizzle], [-]src1[_abs][.swizzle]

Item	Description
<i>dest</i>	[in] The address of the result of the operation.
<i>src0</i>	[in] The components to compare with <i>src1</i> .
<i>src1</i>	[in] The components to compare with <i>src0</i> .

Remarks

This instruction performs the double-precision floating-point comparison ($src0 \neq src1$) for each component, and writes the result to *dest*.

If the comparison is true, then 32-bit 0xFFFFFFFF is returned for that component. Otherwise 32-bit 0x00000000 is returned.

Comparison with NaN returns true.

The valid *dest* masks are any one or two components. That is: .x, .y, .z, .w, .xy, .xz, .xw, .yz, .yw, .zw. The first *dest* component in the mask receives the 32-bit result for the first double comparison. The second component in the mask, if present, receives the 32-bit result for the second double comparison.

The valid swizzles for the source parameters are .xyzw, .xyxy, .zwxy, .zwzw. The following *src* mappings are post-swizzle:

- *src0* is a double vec2 across (x 32LSB, y 32MSB) and (z 32LSB, w 32MSB).
- *src1* is a double vec2 across (x 32LSB, y 32MSB) and (z 32LSB, w 32MSB).

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

drcp (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Computes a component-wise double-precision reciprocal.

drcp[_sat] dest[.mask], [-]src0[.abs][.swizzle]

Item	Description
<i>dest</i>	[in] The address of the results $dest = 1.0 / src0$. The result value must be accurate to 1.0 ULP
<i>src0</i>	[in] The number to take the reciprocal of.

Remarks

The DRCP instruction is emitted by the HLSL compiler only when explicitly called via the `rcp()` intrinsic, when a double is used as the argument. The accuracy of this instruction is required to be 1.0 ULP.

Shaders that use this instruction will be marked with a shader flag that will cause them to fail to bind unless all the following conditions are met.

- The system supports DirectX 11.1.
- The system includes a WDDM 1.2 driver.
- The driver reports support for this instruction via `D3D11_FEATURE_DATA_D3D11_OPTIONS.ExtendedDoublesShaderInstructions` set to `TRUE`.

The following table shows the results obtained when executing the instruction with various classes of numbers, assuming that neither overflow or underflow occurs.

In this table F means finite-real number.

<i>src-></i>	-inf	-F	-0	+0	+F	+inf	NaN
<i>dest-></i>	-0	-F	-inf	+inf	+F	+0	NaN

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

dtof (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Component-wise conversion from double-precision floating-point data to single-precision floating-point data.

```
dtof dest[.mask], [-]src0[.swizzle],
```

Item	Description
<i>dest</i>	[in] The address of the converted data.
<i>src0</i>	[in] The data to be converted.

Remarks

Each component of the source is converted from the double-precision representation to the single-precision representation using round-to-nearest-even rounding.

The valid swizzles for the source parameter are .xyzw, .xyxy, .zwxy, .zwzw.

The valid *dest* masks are any one or two components. That is: .x, .y, .z, .w, .xy, .xz, .xw, .yz, .yw, .zw. The result of the first conversion goes to the first component in the mask, and the result of the second component goes in the second component in the mask, if present.

dest components are float32.

src0 is a double vec2 across (x 32LSB, y 32MSB) and (z 32LSB, w 32MSB) post swizzle.

For float32<->double conversions, implementations may either honor float32 denorms or may flush them.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

emit_stream (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Emit a vertex to a given stream.

emit_stream streamIndex

Item	Description
<i>streamIndex</i>	[in] The stream index.

Remarks

This instruction causes all declared o# registers for the given stream to be read out of the geometry shader to generate a vertex. After the emit, all data in all output registers for all streams become uninitialized, not just the stream emitted to.

streamIndex must be an immediate value [0..3] for a declared stream.

As multiple **emit_stream** calls are issued, primitives are generated.

Restrictions

- **emit_stream** can appear any number of times in a geometry shader, including within flow control.
- If streams have not been declared, then you must use **emit** instead of **emit_stream**.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
			X		

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes

Shader Model	Supported
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

emitThenCut_stream (sm5 - asm)

Article • 03/09/2021 • 2 minutes to read

Equivalent to an [emit](#) command followed by a [cut](#) command.

emitThenCut_stream streamIndex

Item	Description
<i>streamIndex</i>	[in] The stream index.

Remarks

This operation is useful when knowingly outputting the last vertex in a topology.

If streams have not been declared, then you must use [emitThenCut](#) instead of [emitThenCut_stream](#).

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
			X		

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

f16tof32 (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Component-wise float16 to float32 conversion.

f16tof32 dest[.mask], [-]src0[.swizzle]

Item	Description
<i>dest</i>	[in] The address of the float32 result.
<i>src0</i>	[in] The float16 value to convert.

Remarks

This operation performs a component-wise conversion of a float16 value from LSB bits to a float32 result.

This operation follows D3D rules for floating point conversion.

Use this instruction for shader-driven data decompression.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no

Shader Model	Supported
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

f32tof16 (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Component-wise float16 to float32 conversion.

f32tof16 dest[.mask], [-]src0[.swizzle]

Item	Description
<i>dest</i>	[in] The address of float16 result.
<i>src0</i>	[in] The float32 value to convert.

Remarks

This instruction performs a component-wise conversion of a float32 value to a float16 value result placed in LSB 16 bits.

This instruction follows D3D rules for floating point conversion.

Use this instruction for shader-driven data compression.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no

Shader Model	Supported
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

fcall (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Interface function call.

fcall fp#[arrayIndex][callSite]

Item	Description
<i>fp#</i>	[in] The function pointer.
<i>arrayIndex</i>	[in] Optional. Specifies an offset into the function pointer array. This parameter must be a literal unsigned integer if <i>fp#</i> was not declared as indexable. Otherwise, <i>arrayIndex</i> may be of the form literal base + offset from a shader register. For example, fcall fp1[r1.w + 0][0] .
<i>callSite</i>	[in] Optional. A literal unsigned integer offset into the selected function table, selecting a function body <i>fb#</i> to execute.

Remarks

fp#[arrayIndex]() resolves to a particular function table, selected from the API outside the shader from the function table choices listed in the declaration of *fp#*.

The sum of # in *fp#* and *arrayIndex* select the function table. For example, if an interface is declared as fp4[4][3] (array size of 4), the following **fcalls** are equivalent: fcall fp4[2][3] and fp5[1][3], because 4+2 = 5+1.

Restrictions

- If *arrayIndex* uses dynamic indexing, behavior is undefined if *arrayIndex* diverges on adjacent shader invocations, which could be executing in lockstep. The HLSL compiler will attempt to disallow this case.

Adjacent invocations can be inactive due to flow control, because it doesn't break lockstep execution.
- If *fp#* + *arrayIndex* specifies an out of bounds index, behavior is undefined.
- For the undefined cases described here, it means the behavior of the current D3D device becomes undefined, including the possibility of Device Lost. However no memory outside the current D3D device will be accessed or executed as code.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

firstbit (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Finds the first bit set in a number, either from LSB or MSB.

```
firstbit{_hi|_lo|_shi} dest[.mask], src0[.swizzle]
```

Item	Description
<i>dest</i>	[in] The integer position of the first bit set in <i>src0</i> starting from the LSB for <i>firstbit_lo</i> or MSB for <i>firstbit_hi</i> .
<i>src0</i>	[in] The input integer.

Remarks

This operation returns the integer position of the first bit set in the 32-bit input starting from the LSB for *firstbit_lo* or MSB for *firstbit_hi*. For example *firstbit_lo* on 0x00000001 returns 0. *firstbit_hi* on 0x10000000 returns 3.

firstbit_shi (s for signed) returns the first 0 from the MSB if the number is negative; otherwise it returns the first 1 from the MSB.

All variants of the instruction return ~0 (0xffffffff in 32-bit register) if no match is found.

Use this instruction to quickly enumerate set bits in a bitfield, or find the largest power of 2 in a number.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Mimimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no

Shader Model	Supported
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

ftod (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Component-wise conversion from single-precision floating-point data to double-precision floating-point data.

```
ftod dest[.mask], [-]src0[.swizzle],
```

Item	Description
<i>dest</i>	[in] The address of the converted data.
<i>src0</i>	[in] The data to be converted.

Remarks

Each component of the source is converted from the single-precision representation to the double-precision representation.

The valid *dest* masks are .xy, .zw, and .xyzw. .xy receives the result of the first conversion, and .zw receives the result of the second conversion.

dest is a double vec2 across (x 32LSB, y 32MSB) and (z 32LSB, w 32MSB).

src0 is a float vec2 across x and y (zw ignored) (post swizzle).

For float32<->double conversions, implementations may either honor float32 denorms or may flush them.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes

Shader Model	Supported
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

gather4 (sm5 - asm)

Article • 03/09/2021 • 2 minutes to read

Gathers the four texels that would be used in a bi-linear filtering operation and packs them into a single register. This instruction only works with 2D or CubeMap textures, including arrays. Only the addressing modes of the sampler are used and the top level of any mip pyramid is used.

```
gather4[_aoffimmi(u,v)] dest[.mask], srcAddress[.swizzle], srcResource[.swizzle],  
srcSampler[.select_component]
```

Item	Description
<i>dest</i>	[in] The address of the results of the operation.
<i>srcAddress</i>	[in] A set of texture coordinates.
<i>srcResource</i>	[in] A texture register.
<i>srcSampler</i>	[in] A sampler register.

Remarks

This instruction behaves like the [sample](#) instruction, but a filtered sample is not generated. The four samples that would contribute to filtering are placed into xyzw in counter clockwise order starting with the sample to the lower left of the queried location. This is the same as point sampling with (u,v) texture coordinate deltas at the following locations: (-,+),(+,+),(+,-),(-,-), where the magnitude of the deltas are always half a texel.

For CubeMap textures, when a bi-linear footprint spans an edge, texels from the neighboring face are used. Corners use the same rules as the [sample](#) instruction; that is, the unknown corner is considered the average of the three impinging face corners.

There are texture format restrictions that apply to **gather4** which are expressed in the format list.

The swizzle on *srcResource* allows the returned values to be swizzled arbitrarily before they are written to the destination.

The *.select_component* on *srcSampler* chooses which component of the source texture (r/g/b/a) to read 4 texels from.

For formats with float32 components, if the value being fetched is normalized, denormalized, +0 or +-INF, it is returned to the shader unaltered. NaN is returned as NaN, but the exact bit representation of the NaN may be changed. For TextureCubes, some synthesis of the missing 4th texel must occur at corners, so returning bits unchanged for the synthesized texel does not apply, and denorms could be flushed.

For hardware implementations, optimizations in traditional bilinear filtering that detect samples directly on texels and skip reading of texels that would have weight 0 cannot be leveraged with this instruction. *gather4* always returns all requested texels.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

gather4_c (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Same as [gather4](#), except this instruction performs comparison on texels, similar to [sample_c](#).

```
gather4_c[_aoffimmi(u,v)] dest[.mask], srcAddress[.swizzle], srcResource[.swizzle],  
srcSampler[.r], srcReferenceValue
```

Item	Description
<i>dest</i>	[in] The address of the result of the operation
<i>srcAddress</i>	[in] A set of texture coordinates.
<i>srcResource</i>	[in] A texture register.
<i>srcSampler</i>	[in] A sampler register.
<i>srcReferenceValue</i>	[in] A register with a single component selected, which is used in the comparison.

Remarks

See [sample_c](#) for a description of how *srcReferenceValue* gets compared against each fetched texel. Unlike [sample_c](#), [gather4_c](#) returns each comparison result, rather than filtering them.

For TextureCube corners, where there are three real texels and a fourth must be synthesized, the synthesis must occur after the comparison step. This means the returned comparison result for the synthesized texel can be 0, 0.33 , 0.66 , or 1. Some implementations may only return either 0 or 1 for the synthesized texel. Aside from this listing of possible results, the method for synthesizing the texel is unspecified.

For formats with float32 components, if the value being fetched is normalized, or +-INF, it is used in the comparison operation untouched. NaN is used in the comparison operation as NaN, but the exact bit representation of the NaN may be changed. Denorms are flushed to zero going into the comparison. For TextureCubes, some synthesis of the missing 4th texel must occur at corners, so the notion of returning bits unchanged for the synthesized texel does not apply.

Formats supported for *gather4_c* are same as those supported for *sample_c*. These are single-component formats, thus the .R on *srcSampler*, rather than an arbitrary swizzle.

`gather4_c` on an unbound resource returns 0.

Use this instruction for custom shadow map filtering.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

gather4_po (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

A variant of [gather4](#), but instead of supporting an immediate offset [-8..7], the offset comes as a parameter to the instruction, and also has larger range of [-32..31].

```
gather4_po dest[.mask], srcAddress[.swizzle], srcOffset[.swizzle], srcResource[.swizzle],  
srcSampler[.select_component]
```

Item	Description
<i>dest</i>	[in] The address of the result of the operation.
<i>srcAddress</i>	[in] A set of texture coordinates.
<i>srcOffset</i>	[in] The offset.
<i>srcResource</i>	[in] A texture register.
<i>srcSampler</i>	[in] A sampler register.

Remarks

The first two components of the 4-vector offset parameter supply 32-bit integer offsets. The other components of this parameter are ignored.

The 6 least significant bits of each offset value is honored as a signed value, yielding [-32..31] range.

This instruction only works with 2D textures, unlike [gather4](#), which also works with TextureCubes.

The only modes honored in the sampler are the addressing modes. Only the most detailed mip in the resource view is used.

If the address falls on a texel center, this does not mean the other texels can be zeroed out.

The *srcSampler* parameter includes [.select_component], allowing any single component of a texture to be retrieved, including returning defaults for missing components.

For formats with float32 components, if the value being fetched is normalized, denormalized, +-0 or +-INF, it is returned to the shader unaltered. NaN is returned as NaN, but the exact bit representation of the NaN may be changed. For TextureCubes,

some synthesis of the missing 4th texel must occur at corners, so the notion of returning bits unchanged for the synthesized texel does not apply, and denorms could be flushed.

Use this instruction to extend offset range of `gather4` to be larger and programmable. The "po" suffix on the name means "programmable offset".

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

gather4_po_c (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Behaves the same as [gather4_po](#), except performs comparison on texels, similar to [sample_c](#).

```
gather4_po_c dest[.mask], srcAddress[.swizzle], srcOffset[.swizzle], srcResource[.swizzle],
srcSampler[.r], srcReferenceValue
```

Item	Description
<i>dest</i>	[in] The address of the result of the operation.
<i>srcAddress</i>	[in] A set of texture coordinates.
<i>srcOffset</i>	[in] The offset.
<i>srcResource</i>	[in] A texture register.
<i>srcSampler</i>	[in] A sampler register.
<i>srcReferenceValue</i>	[in] Single component selected.

Remarks

See [sample_c](#) for information about how *srcReferenceValue* is compared against each fetched texel. Unlike [sample_c](#), *gather4_po_c* returns each comparison result, rather than filtering them.

This instruction, like [gather4_po](#), only works with 2D textures. This is unlike [gather4_c](#), which also works with TextureCubes.

For formats with float32 components, if the value being fetched is normalized, or +-INF, it is used in the comparison operation untouched. NaN is used in the comparison operation as NaN, but the exact bit representation of the NaN may be changed. Denorms are flushed to zero going into the comparison. For TextureCubes, some synthesis of the missing 4th texel must occur at corners, so the notion of returning bits unchanged for the synthesized texel does not apply.

Formats supported for *gather4_po_c* are same as those supported for [sample_c](#). These are single-component formats, thus the .R on *srcSampler*, rather than an arbitrary swizzle.

gather4_po_c on an unbound resource returns 0.

Use this method for shadow map filtering.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

hs_control_point_phase (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Start the control point phase in a hull shader.

hs_control_point_phase

Remarks

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
	X				

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

hs_decls (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Start the declarations phase in a hull shader.

```
hs_decls
```

Remarks

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
	X				

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

hs_fork_phase (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Start the fork phase in a hull shader.

hs_fork_phase

Remarks

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
	X				

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

hs_join_phase (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Start the join phase in a hull shader.

hs_join_phase

Remarks

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
	X				

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

ibfe (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Given a range of bits in a number, shift those bits to the LSB and sign extend the MSB of the range.

ibfe dest[.mask], src0[.swizzle], src1[.swizzle], src2[.swizzle]

Item	Description
<i>dest</i>	[in] The address of the results of the operation.
<i>src0</i>	[in] The bitfield width.
<i>src1</i>	[in] The bitfield offset.
<i>src2</i>	[in] The value to shift.

Remarks

The LSB 5 bits of src0 provide the bitfield width (0-31).

The LSB 5 bits of src1 provide the bitfield offset (0-31).

The following example shows how to use this instruction.

syntax

```
Given width, offset:  
    if( width == 0 )  
    {  
        dest = 0  
    }  
    else if( width + offset < 32 )  
    {  
        shl dest, src2, 32-(width+offset)  
        ishr dest, dest, 32-width  
    }  
    else  
    {  
        ishr dest, src2, offset  
    }
```

Use this instruction to unpack signed integers or flags.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

imm_atomic_alloc (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Atomically increment the hidden 32-bit counter stored with a Count or Append unordered access view (UAV), returning the original value.

```
imm_atomic_alloc dest[.single_component_mask], dstUAV
```

Item	Description
<i>dest</i>	[in] Contains the returned counter value.
<i>dstUAV</i>	[in] A Structured Buffer UAV with the Count or Append flag.

Remarks

There is a hidden unsigned 32-bit integer counter value associated with each Count or Append Buffer view which is initialized when the view is bound to the pipeline, including the option to keep the previous value.

This instruction does an atomic increment of the counter value, returning the original to *dest*.

For an Append UAV, the returned value is only valid for the duration of the shader invocation. after that the implementation may rearrange the memory layout. Any memory addressing based on the returned value must be limited to the shader invocation.

For an Append UAV, within the shader invocation the HLSL compiler can use the returned value as the struct index to use for accessing the structured buffer. Accessing any struct index other than those locations returned by call(s) to `imm_atomic_alloc` or `_consume` produce undefined results in that exactly which memory location within the UAV is being accessed is random and only fixed for the lifetime of the shader invocation.

For a Count UAV, the returned value can be saved by the application as a reference to a fixed location within the UAV that is meaningful after the shader invocation is over. Any location in a Count UAV may always be accessed independent of what the count value is.

There is no clamping of the count, so it wraps on overflow.

The same shader cannot attempt both `imm_atomic_alloc` and `imm_atomic_consume` on the same UAV. Further, the GPU cannot allow multiple shader invocations to mix `imm_atomic_alloc` and `imm_atomic_consume` on the same UAV.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

Because UAVs are available at all shader stages for Direct3D 11.1, this instruction applies to all shader stages for the Direct3D 11.1 runtime, which is available starting with Windows 8.

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

imm_atomic_and (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Immediate atomic bitwise AND to memory. Returns the value in memory before the AND.

```
imm_atomic_and dst0[.single_component_mask], dst1, dstAddress[.swizzle],  
src0[.select_component]
```

Item	Description
<i>dst0</i>	[in] Contains value from <i>dst1</i> before the AND.
<i>dst1</i>	[in] An unordered access view (UAV) (u#). In the compute shader this can also be thread group shared memory (g#).
<i>dstAddress</i>	[in] The destination memory.
<i>src0</i>	The value to AND with <i>dst</i> .

Remarks

This instruction performs a single component 32-bit bitwise AND of operand *src0* with *dst1* at 32-bit per component address *dstAddress*.

If *dst1* is a u#, it may have been declared as raw, typed or structured. If typed, it must be declared as UINT/SINT with the bound resource format being R32_UINT/_SINT.

If *dst1* is g#, it must be declared as raw or structured.

The value in *dst1* memory before the AND is returned to *dst0*.

The entire operation is performed atomically.

The number of components taken from the address is determined by the dimensionality of the resource declared at *dst1*.

If the shader invocation is inactive, for example if the pixel has been discarded earlier in its execution, or a pixel/sample invocation only exists to serve as a helper to a real pixel/sample for derivatives, this instruction does not alter the *dst1* memory at all, and the returned value is undefined.

Out of bounds addressing on u# causes nothing to be written to memory, except if the u# is structured, and byte offset into the struct (second component of the address) is

causing the out of bounds access, then the entire contents of the UAV become undefined.

Out of bounds addressing on `u#` or `g#` causes an undefined result to be returned to the shader in `dst0`.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

Because UAVs are available at all shader stages for Direct3D 11.1, this instruction applies to all shader stages for the Direct3D 11.1 runtime, which is available starting with Windows 8.

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

imm_atomic_cmp_exch (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Immediate compare and exchange to memory.

```
imm_atomic_cmp_exch dst0[.single_component_mask], dst1, dstAddress[.swizzle],  
src0[.select_component], src1[.select_component]
```

Item	Description
<i>dst0</i>	[out] Contains <i>dst1</i> before the write.
<i>dst1</i>	[in] An unordered access view (UAV) (u#). In the compute shader this can also be thread group shared memory (g#).
<i>dstAddress</i>	[in] The destination memory.
<i>src0</i>	[in] The value to compare to <i>dst1</i> .
<i>src1</i>	[in] The value written to the destination memory if the compared values are identical.

This instruction performs a single component 32-bit value compare of operand *src0* with *dst1* at 32-bit per component address *dstAddress*.

If *dst1* is a u#, it may have been declared as raw, typed or structured. If typed, it must be declared as UINT/SINT with the bound resource format being R32_UINT/_SINT.

If *dst1* is g#, it must be declared as raw or structured.

If the compared values are identical, the single-component 32-bit value in *src1* is written to the destination memory. Otherwise, the destination memory is not changed.

The original 32-bit value in the destination memory is always written to *dst0*.

The entire operation is performed atomically.

If the shader invocation is inactive, for example if the pixel has been discarded earlier in its execution, or a pixel/sample invocation only exists to serve as a helper to a real pixel/sample for derivatives, this instruction does not alter the *dst1* memory at all, and the returned value is undefined.

Out of bounds addressing on u# causes nothing to be written to memory, except if the u# is structured, and byte offset into the struct (second component of the address) is causing the out of bounds access, then the entire contents of the UAV become undefined.

Out of bounds addressing on u# or g# causes an undefined result to be returned to the shader in *dst0*.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

Because UAVs are available at all shader stages for Direct3D 11.1, this instruction applies to all shader stages for the Direct3D 11.1 runtime, which is available starting with Windows 8.

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

imm_atomic_consume (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Atomically decrement the hidden 32-bit counter stored with a Count or Append unordered access view (UAV), returning the new value.

imm_atomic_consume dest[.single_component_mask], dstUAV

Item	Description
<i>dest</i>	[in] Contains the returned original counter value.
<i>dstUAV</i>	[in] A structured buffer UAV with the Count or Append flag.

Remarks

See [imm_atomic_alloc](#) for a discussion about the validity of the returned count value depending on whether the UAV is Count or Append. The same applies for [imm_atomic_consume](#).

imm_atomic_consume does an atomic decrement of the counter value, returning the new value to *dest*.

There is no clamping of the count, so it wraps on underflow.

The same shader cannot attempt both [imm_atomic_alloc](#) and [imm_atomic_consume](#) on the same UAV. Further, the GPU cannot allow multiple shader invocations to mix [imm_atomic_alloc](#) and [imm_atomic_consume](#) on the same UAV.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

Because UAVs are available at all shader stages for Direct3D 11.1, this instruction applies to all shader stages for the Direct3D 11.1 runtime, which is available starting with Windows 8.

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

imm_atomic_exch (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Immediate atomic exchange to memory.

```
imm_atomic_exch dst0[.single_component_mask], dst1, dstAddress[.swizzle],  
src0[.select_component]
```

Item	Description
<i>dst0</i>	[in] Contains the value from <i>dst1</i> before the write.
<i>dst1</i>	[in] An unordered access view (UAV) (u#). In the Compute Shader this can also be Thread Group Shared Memory (g#).
<i>dstAddress</i>	[in] The memory address.
<i>src0</i>	[in] The value to write to <i>dst1</i> at <i>dstAddress</i> .

Remarks

This instruction performs a single component 32-bit value write of operand *src0* to *dst1* at 32-bit per component address *dstAddress*.

If *dst1* is a u#, it may have been declared as raw, typed or structured. If typed, it must be declared as UINT/SINT with the bound resource format being R32_UINT/_SINT.

If *dst1* is g#, it must be declared as raw or structured.

The number of components taken from the address is determined by the dimensionality of the resource declared at *dst1*.

The original 32-bit value in the destination memory is written to *dst0*.

The entire operation is performed atomically.

If the shader invocation is inactive, for example if the pixel has been discarded earlier in its execution, or a pixel/sample invocation only exists to serve as a helper to a real pixel/sample for derivatives, this instruction does not alter the *dst1* memory at all, and the returned value is undefined.

Out of bounds addressing on u# causes nothing to be written to memory, except if the u# is structured, and byte offset into the struct (second component of the address) is

causing the out of bounds access, then the entire contents of the UAV become undefined.

Out of bounds addressing on `u#` or `g#` causes an undefined result to be returned to the shader in `dst0`.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

Because UAVs are available at all shader stages for Direct3D 11.1, this instruction applies to all shader stages for the Direct3D 11.1 runtime, which is available starting with Windows 8.

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

imm_atomic_iadd (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Immediate atomic integer add to memory. Returns the value in memory before the add.

```
imm_atomic_iadd dst0[.single_component_mask], dst1, dstAddress[.swizzle],  
src0[.select_component]
```

Item	Description
<i>dst0</i>	[in] Contains the value in <i>dst1</i> before the write.
<i>dst1</i>	This value must be an unordered access view (UAV) (u#). In the compute shader it can also be thread group shared memory (g#).
<i>dstAddress</i>	[in] The memory naddress.
<i>src0</i>	[in] The value to add to <i>dst1</i> .

Remarks

This instruction performs a single component 32-bit integer add of operand *src0* with *dst1* at 32-bit per component address *dstAddress*. It is insensitive to sign.

If *dst1* is a u#, it may have been declared as raw, typed or structured. If typed, it must be declared as UINT/SINT with the bound resource format being R32_UINT/_SINT.

If *dst1* is g#, it must be declared as raw or structured.

The value in *dst1* memory before addition is returned to *dst0*.

The number of components taken from the address is determined by the dimensionality of *dst1*.

The entire operation is performed atomically.

If the shader invocation is inactive, for example if the pixel has been discarded earlier in its execution, or a pixel/sample invocation only exists to serve as a helper to a real pixel/sample for derivatives, this instruction does not alter the *dst1* memory at all, and the returned value is undefined.

Out of bounds addressing on u# causes nothing to be written to memory, except if the u# is structured, and byte offset into the struct (second component of the address) is

causing the out of bounds access, then the entire contents of the UAV become undefined.

Out of bounds addressing on `u#` or `g#` causes an undefined result to be returned to the shader in `dst0`.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

Because UAVs are available at all shader stages for Direct3D 11.1, this instruction applies to all shader stages for the Direct3D 11.1 runtime, which is available starting with Windows 8.

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

imm_atomic_imax (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Immediate atomic signed max to memory. Returns value in memory before the max operation.

```
imm_atomic_imax dst0[.single_component_mask], dst1, dstAddress[.swizzle],  
src0[.select_component]
```

Item	Description
<i>dst0</i>	[in] Contains the value from <i>dst1</i> before this instruction.
<i>dst1</i>	[in] An unordered access view (UAV) (u#). In the compute shader this can also be thread group shared memory (g#).
<i>dstAddress</i>	[in] The memory address.
<i>src0</i>	[in] The value to compare to <i>dst1</i> at <i>dstAddress</i> .

Remarks

This instruction performs a single component 32-bit signed maximum of operand *src0* with *dst1* at 32-bit per component address *dstAddress*.

If *dst1* is a u#, it may have been declared as raw, typed or structured. If typed, it must be declared as UINT/SINT with the bound resource format being R32_UINT/_SINT.

If *dst1* is g#, it must be declared as raw or structured.

The value in *dst1* memory before max is returned to *dst0*.

The number of components taken from the address is determined by the dimensionality of *dst1*.

The entire operation is performed atomically.

If the shader invocation is inactive, for example if the pixel has been discarded earlier in its execution, or a pixel/sample invocation only exists to serve as a helper to a real pixel/sample for derivatives, this instruction does not alter the *dst1* memory at all, and the returned value is undefined.

Out of bounds addressing on u# causes nothing to be written to memory, except if the u# is structured, and byte offset into the struct (second component of the address) is

causing the out of bounds access, then the entire contents of the UAV become undefined.

Out of bounds addressing on `u#` or `g#` causes an undefined result to be returned to the shader in `dst0`.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

Because UAVs are available at all shader stages for Direct3D 11.1, this instruction applies to all shader stages for the Direct3D 11.1 runtime, which is available starting with Windows 8.

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

imm_atomic_imin (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Immediate atomic signed min to memory. Returns value in memory before the max operation.

```
imm_atomic_imin dst0[.single_component_mask], dst1, dstAddress[.swizzle],  
src0[.select_component]
```

Item	Description
<i>dst0</i>	[in] Contains the value from <i>dst1</i> before this instruction.
<i>dst1</i>	[in] An unordered access view (UAV) (u#). In the compute shader this can also be thread group shared memory (g#).
<i>dstAddress</i>	[in] The memory address.
<i>src0</i>	[in] The value to compare to <i>dst1</i> at <i>dstAddress</i> .

Remarks

This instruction performs a single component 32-bit signed minimum of operand *src0* with *dst1* at 32-bit per component address *dstAddress*.

If *dst1* is a u#, it may have been declared as raw, typed or structured. If typed, it must be declared as UINT/SINT with the bound resource format being R32_UINT/_SINT.

If *dst1* is g#, it must be declared as raw or structured.

The value in *dst1* memory before min is returned to *dst0*.

The number of components taken from the address is determined by the dimensionality of *dst1*.

The entire operation is performed atomically.

If the shader invocation is inactive, for example if the pixel has been discarded earlier in its execution, or a pixel/sample invocation only exists to serve as a helper to a real pixel/sample for derivatives, this instruction does not alter the *dst1* memory at all, and the returned value is undefined.

Out of bounds addressing on u# causes nothing to be written to memory, except if the u# is structured, and byte offset into the struct (second component of the address) is

causing the out of bounds access, then the entire contents of the UAV become undefined.

Out of bounds addressing on `u#` or `g#` causes an undefined result to be returned to the shader in `dst0`.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

Because UAVs are available at all shader stages for Direct3D 11.1, this instruction applies to all shader stages for the Direct3D 11.1 runtime, which is available starting with Windows 8.

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

imm_atomic_or (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Immediate atomic bitwise OR to memory. Returns the value in memory before the OR.

```
imm_atomic_or dst0[.single_component_mask], dst1, dstAddress[.swizzle],  
src0[.select_component]
```

Item	Description
<i>dst0</i>	[in] Contains the value from <i>dst1</i> before the OR.
<i>dst1</i>	[in] An unordered access view (UAV) (u#). In the compute shader this can also be thread group shared memory (g#).
<i>dstAddress</i>	[in] The memory address.
<i>src0</i>	The value to OR with <i>dst1</i> .

Remarks

This instruction performs a single component 32-bit bitwise OR of operand *src0* with *dst1* at 32-bit per component address *dstAddress*.

If *dst1* is a u#, it may have been declared as raw, typed or structured. If typed, it must be declared as UINT/SINT with the bound resource format being R32_UINT/_SINT.

If *dst1* is g#, it must be declared as raw or structured.

The value in *dst1* memory before the OR is returned to *dst0*.

The entire operation is performed atomically.

The number of components taken from the address is determined by the dimensionality of the resource declared at *dst1*.

If the shader invocation is inactive, for example if the pixel has been discarded earlier in its execution, or a pixel/sample invocation only exists to serve as a helper to a real pixel/sample for derivatives, this instruction does not alter the *dst1* memory at all, and the returned value is undefined.

Out of bounds addressing on u# causes nothing to be written to memory, except if the u# is structured, and byte offset into the struct (second component of the address) is

causing the out of bounds access, then the entire contents of the UAV become undefined.

Out of bounds addressing on `u#` or `g#` causes an undefined result to be returned to the shader in `dst0`.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

Because UAVs are available at all shader stages for Direct3D 11.1, this instruction applies to all shader stages for the Direct3D 11.1 runtime, which is available starting with Windows 8.

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

imm_atomic_umax (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Immediate atomic unsigned max to memory. Returns value in memory before the max operation.

```
imm_atomic_umax dst0[.single_component_mask], dst1, dstAddress[.swizzle],  
src0[.select_component]
```

Item	Description
<i>dst0</i>	[in] Contains the value from <i>dst1</i> before this instruction.
<i>dst1</i>	[in] An unordered access view (UAV) (u#). In the compute shader this can also be thread group shared memory (g#).
<i>dstAddress</i>	[in] The memory address.
<i>src0</i>	[in] The value to compare to <i>dst1</i> at <i>dstAddress</i> .

Remarks

This instruction performs a single component 32-bit unsigned maximum of operand *src0* with *dst1* at 32-bit per component address *dstAddress*.

If *dst1* is a u#, it may have been declared as raw, typed or structured. If typed, it must be declared as UINT/SINT with the bound resource format being R32_UINT/_SINT.

If *dst1* is g#, it must be declared as raw or structured.

The value in *dst1* memory before max is returned to *dst0*.

The number of components taken from the address is determined by the dimensionality of *dst1*.

The entire operation is performed atomically.

If the shader invocation is inactive, for example if the pixel has been discarded earlier in its execution, or a pixel/sample invocation only exists to serve as a helper to a real pixel/sample for derivatives, this instruction does not alter the *dst1* memory at all, and the returned value is undefined.

Out of bounds addressing on u# causes nothing to be written to memory, except if the u# is structured, and byte offset into the struct (second component of the address) is

causing the out of bounds access, then the entire contents of the UAV become undefined.

Out of bounds addressing on `u#` or `g#` causes an undefined result to be returned to the shader in `dst0`.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

Because UAVs are available at all shader stages for Direct3D 11.1, this instruction applies to all shader stages for the Direct3D 11.1 runtime, which is available starting with Windows 8.

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

imm_atomic_umin (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Immediate atomic unsigned min to memory. Returns value in memory before the max operation.

```
imm_atomic_umin dst0[.single_component_mask], dst1, dstAddress[.swizzle],  
src0[.select_component]
```

Item	Description
<i>dst0</i>	[in] Contains the value from <i>dst1</i> before this instruction.
<i>dst1</i>	[in] An unordered access view (UAV) (u#). In the compute shader this can also be thread group shared memory (g#).
<i>dstAddress</i>	[in] The memory address.
<i>src0</i>	[in] The value to compare to <i>dst1</i> at <i>dstAddress</i> .

Remarks

This instruction performs a single component 32-bit unsigned minimum of operand *src0* with *dst1* at 32-bit per component address *dstAddress*.

If *dst1* is a u#, it may have been declared as raw, typed or structured. If typed, it must be declared as UINT/SINT with the bound resource format being R32_UINT/_SINT.

If *dst1* is g#, it must be declared as raw or structured.

The value in *dst1* memory before min is returned to *dst0*.

The number of components taken from the address is determined by the dimensionality of *dst1*.

The entire operation is performed atomically.

If the shader invocation is inactive, for example if the pixel having been discarded earlier in its execution, or a pixel/sample invocation only exists to serve as a helper to a real pixel/sample for derivatives, this instruction does not alter the *dst1* memory at all, and the returned value is undefined.

Out of bounds addressing on u# causes nothing to be written to memory, except if the u# is structured, and byte offset into the struct (second component of the address) is

causing the out of bounds access, then the entire contents of the UAV become undefined.

Out of bounds addressing on `u#` or `g#` causes an undefined result to be returned to the shader in `dst0`.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

Because UAVs are available at all shader stages for Direct3D 11.1, this instruction applies to all shader stages for the Direct3D 11.1 runtime, which is available starting with Windows 8.

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

imm_atomic_xor (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Immediate atomic bitwise XOR to memory. Returns value in memory before the XOR.

```
imm_atomic_xor dst0[.single_component_mask], dst1, dstAddress[.swizzle],  
src0[.select_component]
```

Item	Description
<i>dst0</i>	[in] Contains the value from <i>dst1</i> before the XOR.
<i>dst1</i>	[in] An unordered access view (UAV) (u#). In the compute shader this can also be thread group shared memory (g#).
<i>dstAddress</i>	[in] The memory address.
<i>src0</i>	The value to XOR with <i>dst1</i> .

Remarks

This instruction performs a single component 32-bit bitwise XOR of operand *src0* with *dst1* at 32-bit per component address *dstAddress*.

If *dst1* is a u#, it may have been declared as raw, typed or structured. If typed, it must be declared as UINT/SINT with the bound resource format being R32_UINT/_SINT.

If *dst1* is g#, it must be declared as raw or structured.

The value in *dst1* memory before the XOR is returned to *dst0*.

The entire operation is performed atomically.

The number of components taken from the address is determined by the dimensionality of the resource declared at *dst1*.

If the shader invocation is inactive, for example if the pixel has been discarded earlier in its execution, or a pixel/sample invocation only exists to serve as a helper to a real pixel/sample for derivatives, this instruction does not alter the *dst1* memory at all, and the returned value is undefined.

Out of bounds addressing on u# causes nothing to be written to memory, except if the u# is structured, and byte offset into the struct (second component of the address) is

causing the out of bounds access, then the entire contents of the UAV become undefined.

Out of bounds addressing on `u#` or `g#` causes an undefined result to be returned to the shader in `dst0`.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

Because UAVs are available at all shader stages for Direct3D 11.1, this instruction applies to all shader stages for the Direct3D 11.1 runtime, which is available starting with Windows 8.

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

ishl (sm5 - asm)

Article • 03/09/2021 • 2 minutes to read

Shift left.

ishl dest[.mask], src0[.swizzle], src1[.swizzle]

Item	Description
<i>dest</i>	[in] Contains the results of the shift.
<i>src0</i>	[in] The 32-bit values to shift.
<i>src1</i>	[in] The number of bits to shift.

Remarks

This instruction performs a component-wise shift of each 32-bit value in *src0* left by an unsigned integer bit count provided by the LSB 5 bits (0-31 range) in *src1*, inserting 0. The 32-bit per component results are placed in *dest*.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

ishr (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Arithmetic shift right (sign extending).

ishr dest[.mask], src0[.swizzle], src1[.swizzle]

Item	Description
<i>dest</i>	[in] Contains the results of the shift.
<i>src0</i>	[in] The number of bits to shift.
<i>src1</i>	[in] The 32-bit values to shift.

Remarks

This instruction performs a component-wise arithmetic shift of each 32-bit value in *src0* right by an unsigned integer bit count provided by the LSB 5 bits (0-31 range) in *src1*, replicating the value of bit 31. The 32-bit per component result is placed in *dest*.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

Id_raw (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Random-access read of 1-4 32bit components from a raw buffer.

Id_raw dest[.mask], srcByteOffset[.select_component], src0[.swizzle]

Item	Description
<i>dest</i>	[in] The address of the result of the operation.
<i>srcByteOffset</i>	[in] Specifies the offset to read from.
<i>src0</i>	[in] The component to read.

Remarks

src0 must be:

- Any shader stage: SRV (t#)Id st
- Compute shader or pixel shader: UAV (u#)
- Compute shader: thread group shared memory (g#)

srcByteOffset specifies the base 32-bit value in memory for a window of 4 sequential 32-bit values in which data may be read, depending on the swizzle and mask on other parameters.

The data read from the raw buffer is equivalent to the following pseudocode: where we have the offset, address, pointer to the buffer contents, stride of the source, and the data stored linearly.

syntax

```
BYTE *BufferContents;           // from src0
UINT srcByteOffset;           // from srcByteOffset
BYTE *ReadLocation;           // value to calculate
ReadLocation = BufferContents
    + srcByteOffset;

UINT32 Temp[4];   // used to make code shorter

// apply the source resource swizzle on source data
Temp = read_and_swizzle(ReadLocation, srcSwizzle);
```

```
// write the components to the output based on mask
ApplyWriteMask(dstRegister, dstWriteMask, Temp);
```

Out of bounds addressing on u#/t# of any given 32-bit component returns 0 for that component.

Out of bounds addressing on g# (the bounds of that particular g#, as opposed to all shared memory) for any given 32-bit component returns an undefined result.

cs_4_0 and cs_4_1 support this instruction for UAV and SRV.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Because UAVs are available at all shader stages for Direct3D 11.1, this instruction applies to all shader stages for UAVs for the Direct3D 11.1 runtime, which is available starting with Windows 8.

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

cs_4_0 and cs_4_1 support this instruction for UAV and SRV.

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

Id_structured (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Random-access read of 1-4 32bit components from a structured buffer.

**Id_structured dest[.mask], srcAddress[.select_component],
srcByteOffset[.select_component], src0[.swizzle]**

Item	Description
<i>dest</i>	[in] The address of the results of the operation.
<i>srcAddress</i>	[in] Specifies the index of the structure to read.
<i>srcByteOffset</i>	[in] Specifies the byte offset in the structure to start reading from.
<i>src0</i>	The buffer to read from. This parameter must be a SRV (t#), UAV (u#). In the compute shader it can also be thread group shared memory (g#).

Remarks

The data read from the structure is equivalent to the following pseudocode: where we have the offset, address, pointer to the buffer contents, stride of the source, and the data stored linearly.

syntax

```
registers
    BYTE *BufferContents;           // from SRV or UAV
    UINT BufferStride;             // from base resource
    UINT srcAddress, srcByteOffset; // from source

    BYTE *ReadLocation;           // value to calculate
    ReadLocation = BufferContents
        + BufferStride * srcAddress
        + srcByteOffset;

    UINT32 Temp[4];   // used to make code shorter

    // apply the source resource swizzle on source data
    Temp = read_and_swizzle(ReadLocation, srcSwizzle);

    // write the components to the output based on mask
    ApplyWriteMask(dstRegister, dstWriteMask, Temp);
```

This pseudocode shows how the operation functions, but the actual data does not have to be stored linearly. If the data is not stored linearly, the actual operation of the

instruction needs to match the behavior of the above operation.

Out of bounds addressing on u#/t# of any given 32-bit component returns 0 for that component, except if *srcByteOffset*, plus swizzle is what causes out of bounds access to u#/t#, the returned value for all component(s) is undefined.

Out of bounds addressing on g# (the bounds of that particular g#, as opposed to all shared memory) for any given 32-bit component returns an undefined result.

srcByteOffset is a separate argument from *srcAddress* because it is commonly a literal. This parameter separation has not been done for atomics on structured memory.

cs_4_0 and cs_4_1 support this instruction for UAV, SRV, and TGSM.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Because UAVs are available at all shader stages for Direct3D 11.1, this instruction applies to all shader stages for UAVs for the Direct3D 11.1 runtime, which is available starting with Windows 8.

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

`cs_4_0` and `cs_4_1` support this instruction for UAV, SRV and TGSM.

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

Id_uav_typed (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Random-access read of an element from a typed unordered access view (UAV).

Id_uav_typed dest[.mask], srcAddress[.swizzle], srcUAV[.swizzle]

Item	Description
<i>dest</i>	[in] The address of the results of the operation.
<i>srcAddress</i>	[in] Specifies the address to read from.
<i>srcUAV</i>	[in] The source to read from.

Remarks

This instruction performs a 4-component element read from *srcUAV* at the unsigned integer address in *srcAddress*, converted to 32bit per component based on the format, then written to *dest* in the shader.

srcUAV is a UAV (u#) declared as typed. However, the type of the bound resource must be R32_UINT/SINT/FLOAT.

The number of 32-bit unsigned integer components taken from the address are determined by the dimensionality of the resource declared at *srcUAV*. Addressing is the same as the [Id](#) instruction.

Out of bounds addressing is the same as the [Id](#) instruction.

The behavior of this instruction is identical to the [Id](#) instruction if called as **Id
dest[.mask], srcAddress[.swizzle], srcUAV[.swizzle]**

It is invalid and undefined to use this instruction on a UAV that is not declared as typed. Doing this on a structured or typeless UAV is invalid.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

Because UAVs are available at all shader stages for Direct3D 11.1, this instruction applies to all shader stages for the Direct3D 11.1 runtime, which is available starting with Windows 8.

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

cs_4_0 and cs_4_1 support this instruction for UAV, SRV and TGSM.

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

rcp (sm5 - asm)

Article • 04/26/2021 • 2 minutes to read

Component-wise reciprocal.

rcp[_sat] dest[.mask], [-]src0[_abs][.swizzle]

Item	Description
<i>dest</i>	[in] The address of the results $dest = 1.0f / src0$.
<i>src0</i>	[in] The number to take the reciprocal of.

Remarks

Use this instruction for reduced precision reciprocal, independent of the strict requirements for divide.

Maximum relative error is 2-21. (The error tolerance just matches rsq)

The following table shows the results obtained when executing the instruction with various classes of numbers.

<i>src</i>	-inf	-F	-denorm	-0	+0	+denorm	+F	+inf	NaN
<i>dest</i>	-0	-F	-inf	-inf	+inf	+inf	+F	+0	NaN

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no

Shader Model	Supported
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

store_raw (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Random-access write of 1-4 32bit components into typeless memory.

store_raw dest[.write_mask], dstByteOffset[.select_component], src0[.swizzle]

Item	Description
<i>dest</i>	[in] The memory address.
<i>dstByteOffset</i>	[in] The offset.
<i>src0</i>	[in] The components to write.

Remarks

This instruction performs 1-4 component *32-bit components written from *src0* to *dest* at the offset in *dstByteOffset*. There is no format conversion.

dest must be a UAV (u#), or in the compute shader it can also be thread group shared memory (g#).

dstByteOffset specifies the base 32-bit value in memory for a window of 4 sequential 32-bit values in which data may be written, depending on the swizzle and mask on other parameters.

The location of the data written is equivalent to the following pseudocode which show the address, pointer to the buffer contents, and the data stored linearly.

Syntax

```
BYTE *BufferContents;           // from src0
UINT dstByteOffset;            // source register
BYTE *WriteLocation;           // value to calculate

// calculate writing location
WriteLocation = BufferContents
    + dstByteOffset;

// calculate the number of components to write
switch (dstWriteMask)
{
    x:    WriteComponents = 1; break;
    xy:   WriteComponents = 2; break;
    xyz:  WriteComponents = 3; break;
```

```

        xyzw: WriteComponents = 4; break;
        default: // only these masks are valid
    }

    // copy the data from the source register with
    // the swizzle applied
    memcpy(WriteLocation, swizzle(src0, src0.swizzle),
           WriteComponents * sizeof(UINT32));

```

This pseudocode shows how the operation functions, but the actual data does not have to be stored linearly. *dest* can only have a write mask that is one of the following: .x, .xy, .xyz, .xyzw. The write mask determines the number of 32bit components to write without gaps.

Out of bounds addressing on u# means nothing is written to the out of bounds memory; any part that is in bounds is written correctly.

Out of bounds addressing on g# (the bounds of that particular g#, as opposed to all shared memory) for any given 32-bit component causes the entire contents of all shared memory to become undefined.

cs_4_0 and cs_4_1 support this instruction for UAV.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

Because UAVs are available at all shader stages for Direct3D 11.1, this instruction applies to all shader stages for the Direct3D 11.1 runtime, which is available starting with Windows 8.

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes

Shader Model	Supported
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

store_structured (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Random-access write of 1-4 32-bit components into a structured buffer unordered access view (UAV).

```
store_structured dest[.write_mask], dstAddress[.select_component],
dstByteOffset[.select_component], src0[.swizzle]
```

Item	Description
<i>dest</i>	[in] The address of the results of the operation.
<i>dstAddress</i>	[in] The address at which to write.
<i>dstByteOffset</i>	[in] The index of the structure to write.
<i>src0</i>	[in] The components to write.

Remarks

This instruction performs 1-4 component *32bit components written from *src0* to *dest* at the address in *dstAddress* and *dstByteOffset*. No format conversion.

dest must be a UAV (u#). In the compute shader it can also be thread group shared memory (g#).

dstAddress specifies the index of the structure to write.

The location of the data written is equivalent to the following pseudocode which shows the offset, address, pointer to the buffer contents, stride of the source, and the data stored linearly.

syntax

```
        BYTE *BufferContents;           // from dest
        UINT BufferStride;            // from dest
        UINT dstAddress, dstByteOffset; // source registers
        BYTE *WriteLocation;          // value to calculate

        // calculate writing location
        WriteLocation = BufferContents
                      + BufferStride * dstAddress
                      + dstByteOffset;

        // calculate the number of components to write
```

```

        switch (dstWriteMask)
        {
            x:    WriteComponents = 1; break;
            xy:   WriteComponents = 2; break;
            xyz:  WriteComponents = 3; break;
            xyzw: WriteComponents = 4; break;
            default: // only these masks are valid
        }

        // copy the data from the source register with
        // the swizzle applied
        memcpy(WriteLocation, swizzle(src0, src0.swizzle),
               WriteComponents * sizeof(INT32));

```

This pseudocode shows how the operation functions, but the actual data does not have to be stored linearly. If the data is not stored linearly, the actual operation of the instruction needs to match the behavior of the above operation.

dest can only have a write mask that is one of the following: .x, .xy, .xyz, .xyzw. The write mask determines the number of 32-bit components to write without gaps.

Out of bounds addressing on u# caused by *dstAddress* means nothing is written to the out of bounds memory.

If the *dstByteOffset*, including *dstWriteMask*, is what causes out of bounds access to u#, the entire contents of the UAV become undefined.

Out of bounds addressing on g# (the bounds of that particular g#, as opposed to all shared memory) for any given 32-bit component causes the entire contents of all shared memory to become undefined.

dstByteOffset is a separate argument from *dstAddress* because it is commonly a literal. This parameter separation has not been done for atomics on structured memory.

cs_4_0 and cs_4_1 support this instruction for UAV and TGSM.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

Because UAVs are available at all shader stages for Direct3D 11.1, this instruction applies to all shader stages for the Direct3D 11.1 runtime, which is available starting with Windows 8.

Vertex	Hull	Domain	Geometry	Pixel	Compute

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

store_uav_typed (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Random-access write of an element into a typed unordered access view (UAV).

```
store_uav_typed dstUAV.xyzw, dstAddress[.swizzle], src0[.swizzle]
```

Item	Description
<i>dstUAV</i>	[in] Contains the result of the operation.
<i>dstAddress</i>	[in] The address at which to write.
<i>src0</i>	[in] The components to write.

Remarks

This instruction performs a 4 component *32-bit element written from *src0* to *dstUAV* at the address in *dstAddress*. *dstUAV* is a typed UAV (u#).

The format of the UAV determines format conversion.

The number of 32-bit unsigned integer components taken from the address are determined by the dimensionality of the resource declared at *dstUAV*. This address is in elements.

Out of bounds addressing means nothing gets written to memory.

dstUAV always has a .xyzw write mask. All components must be written.

It is invalid and undefined to use this instruction on a UAV that is not declared as typed. That is, doing this on a structured or typeless UAV is invalid.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

Because UAVs are available at all shader stages for Direct3D 11.1, this instruction applies to all shader stages for the Direct3D 11.1 runtime, which is available starting with Windows 8.

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

swapc (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Performs a component-wise conditional swap of the values between two input registers.

```
swapc dst0[.mask], dst1[.mask], src0[.swizzle], src1[.swizzle], src2[.swizzle]
```

Item	Description
<i>dst0</i>	[in] Register with arbitrary nonempty write masks. Must be different than <i>dst1</i> .
<i>dst1</i>	[in] Register with arbitrary nonempty write masks. Must be different than <i>dst0</i> .
<i>src0</i>	[in] Provides 4 conditions. A nonzero integer value means true .
<i>src1</i>	[in] One of the values to be swapped.
<i>src2</i>	[in] One of the values to be swapped.

Remarks

The encoding of this instruction attempts to compactly express multiple parallel conditional swaps of scalars across two 4-component registers, with minor flexibility in the arrangement of the pairs of numbers involved in swapping.

The choice of register and value for *src0*, *src1*, and *src2* are unconstrained in any way, like [movc](#).

The semantics of this instruction can be described by the equivalent operations with the [movc](#) instruction. The worst case is shown in the following example, making sure destination registers are not updated until the end.

syntax

```
swapc dst0[.mask],  
       dst1[.mask],  
       src0[.swizzle],  
       src1[.swizzle],  
       src2[.swizzle]
```

expands to:

```
movc temp[dst0 s mask],  
      src0[.swizzle],  
      src2[.swizzle], src1[.swizzle]
```

```

    movc dst1[.mask],
        src0[.swizzle],
        src1[.swizzle], src2[.swizzle]

    mov dst0.mask, temp

```

You can choose how to tackle the task, if not directly. For example, the same effect can be achieved by a sequence of up to 4 simple scalar conditional swaps, or as above, two vector **movc** instructions, plus any overhead to make sure the source values are not clobbered by earlier operations in the midst of the expansion.

Use this instruction for sorting.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

sync (sm5 - asm)

Article • 11/20/2019 • 4 minutes to read

Thread group sync or memory barrier.

```
sync[_uglobal|_ugroup][_g][_t]
```

Remarks

Sync has options _uglobal, _ugroup, _g and _t.

In the pixel shader, only **sync_uglobal** is allowed.

In the compute shader, (_uglobal or _ugroup*) and/or _g must be specified. _t is optional in addition.

_uglobal

Global u# (UAV) memory fence.

All prior u# memory reads/writes by this thread in program order are made visible to all threads on the entire GPU before any subsequent u# memory accesses by this thread. The entire GPU part of the definition is replaced by a less-than-global scope in one case, described below.

This applies to all UAV memory bound at the current shader stage.

_uglobal is available in the compute shader or pixel shader.

For any bound UAV that has not been declared by the shader as Globally Coherent, the _uglobal u# memory fence only has visibility within the current compute shader thread-group for that UAV, as if it is _ugroup instead of _uglobal. This issue only applies to the compute shader, since the pixel shader must declare all UAVs as Globally Coherent.

_ugroup

Thread group scope u# (UAV) memory fence.

All prior u# memory reads or writes by this thread in program order are made visible to all threads in the thread group before any subsequent u# memory accesses by this

thread.

This applies to all UAV memory bound at the current shader stage.

_ugroup is available in the compute shader only.

If _ugroup is exposed, for some implementations. The advantage of specifying _ugroup instead of _uglobal is that the **sync** operation can complete more quickly.

Other implementations do not distinguish _ugroup from _uglobal, so both operations are equivalent and behave like _uglobal. Applications can specify their intent by requesting the narrowest scope of **sync** necessary.

Even if a particular UAV is declared as Globally Coherent, a _ugroup **sync** operation will function more efficiently on that UAV if a global barrier is not required.

_g

g# (thread group shared memory) fence.

All prior g# memory reads or writes by this thread in program order are made visible to all threads in the thread group before any subsequent g# memory accesses by this thread.

This applies to all of the current thread group's g# shared memory.

_g is available in the compute shader only.

_t

Thread group sync. All threads within a single thread group (those that can share access to a common set of shared register space) will be executed up to the point where they reach this instruction before any thread can continue.

_t cannot be placed in dynamic flow control, (branches which could vary within a thread group), but can be present in uniform flow control, where all threads in the group pick the same path.

_t is available in the compute shader only.

The following is a listing of compute shader 'sync' variants.

- sync_g
- sync_ugroup*
- sync_uglobal

- sync_g_t
- sync_ugroup_t*
- sync_uglobal_t
- sync_ugroup_g*
- sync_uglobal_g
- sync_ugroup_g_t*
- sync_uglobal_g_t

*Variants with _ugroup may not be targeted by the HLSL compiler, per the earlier discussion in the _ugroup section above.

Listing of pixel shader sync variants include sync_uglobal only.

Memory fences prevent affected instructions from being reordered by compilers or hardware across the fence.

Multiple reads from the same address by a shader invocation that are not separated by memory barriers or writes to the address can be collapsed together. The same applies to writes. Accesses separated by a barrier cannot be merged or moved across the barrier.

Memory fences are not necessary for atomic operations to a given address by different threads to function correctly. Fences are needed when atomics and/or load/store operations need to be synchronized with respect to each other as they appear in individual threads from the point of view of other threads.

In the pixel shader, [discard](#) instructions imply a sync_uglobal fence, in that instructions cannot be reordered across the [discard](#). sync_uglobal in helper pixels (which run only to support derivatives) or discarded pixels may or may not have any affect. It is not allowed for helper or discarded pixels to write to UAVs if, in the case of discard, the writes are issued after the discard. Returned values from UAVs are not allowed to contribute to derivative calculations. Therefore whether or not sync_u is honored for helper pixels or when issued after a discard is moot.

cs_4_0 and cs_4_1 support this instruction.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
				X	X

Because UAVs are available at all shader stages for Direct3D 11.1, the sync_uglobal variant of this instruction applies to all shader stages for the Direct3D 11.1 runtime,

which is available starting with Windows 8.

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

uaddc (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Unsigned integer add with carry.

uaddc dst0[.mask], dst1[.mask], src0[.swizzle], src1[.swizzle]

Item	Description
<i>dst0</i>	[in] Address of the result.
<i>dst1</i>	[in] 1 if carry is produced. Otherwise 0.
<i>src0</i>	[in] 32-bit operand to be added.
<i>src1</i>	[in] 32-bit operand to be added.

Remarks

dst1 can be NULL if the carry is not needed.

Use this instruction for high precision arithmetic.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no

Shader Model	Supported
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

ubfe (sm5 - asm)

Article • 11/20/2019 • 2 minutes to read

Given a range of bits in a number, shift those bits to the LSB and set remaining bits to 0.

ubfe dest[.mask], src0[.swizzle], src1[.swizzle], src2[.swizzle]

Item	Description
<i>dest</i>	[in] Contains the results of the instruction.
<i>src0</i>	[in] The LSB 5 bits provide the bitfield width (0-31).
<i>src1</i>	[in] The LSB 5 bits of <i>src1</i> provide the bitfield offset (0-31).
<i>src2</i>	[in] The number to shift.

Remarks

syntax

```
Given width, offset:  
    if( width == 0 )  
    {  
        dest = 0  
    }  
    else if( width + offset < 32 )  
    {  
        shl dest, src2, 32-(width+offset)  
        ushr dest, dest, 32-width  
    }  
    else  
    {  
        ushr dest, src2, offset  
    }
```

Use this instruction to unpack unsigned integers or flags.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

ushr (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Shift right.

ushr dest[.mask], src0[.swizzle], src1[.swizzle]

Item	Description
<i>dest</i>	[in] Contains the results of the instruction.
<i>src0</i>	[in] The 32-bit values to shift.
<i>src1</i>	[in] The LSB 5 bits provide the number of bits to shift (0-31).

This instruction performs a component-wise shift of each 32-bit value in *src0* right by an unsigned integer bit count provided by the LSB 5 bits (0-31 range) in *src1*, inserting 0. The 32-bit per component results is placed in *dest*.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

Shader Model 5 Assembly (DirectX HLSL)

usubb (sm5 - asm)

Article • 03/08/2023 • 2 minutes to read

Unsigned integer subtract with borrow.

usubb dst0[.mask], dst1[.mask], src0[.swizzle], src1[.swizzle]

Item	Description
<i>dst0</i>	[in] Contains the LSAB results of the instruction.
<i>dst1</i>	[in] The corresponding component of <i>dst0</i> that specifies if a borrow was produced.
<i>src0</i>	[in] The value from which to subtract.
<i>src1</i>	[in] The amount to subtract from <i>src0</i> .

Remarks

This instruction performs a component-wise unsigned subtract of 32-bit operands *src1* from *src0*, placing the LSB part of the 32-bit result in *dst0*.

The corresponding component in *dst1* is written with 1 if a borrow is produced, 0 otherwise.

dst1 can be NULL if the borrow is not needed.

This instruction applies to the following shader stages:

Vertex	Hull	Domain	Geometry	Pixel	Compute
X	X	X	X	X	X

Minimum Shader Model

This instruction is supported in the following shader models:

Shader Model	Supported
Shader Model 5	yes
Shader Model 4.1	no
Shader Model 4	no

Shader Model	Supported
Shader Model 3 (DirectX HLSL)	no
Shader Model 2 (DirectX HLSL)	no
Shader Model 1 (DirectX HLSL)	no

Related topics

[Shader Model 5 Assembly \(DirectX HLSL\)](#)

Software Shaders

Article • 08/23/2019 • 2 minutes to read

Software shaders are implemented to allow development of shaders without underlying hardware support. They support the full feature set. Because they are implemented in software, they will not produce the best performance.

Version	Feature Set	Requirements
vs_2_sw	All the features of vs_2_x	Only supported by software vertex processing and a reference device.
vs_3_sw	All the features of vs_3_0	Only supported by software vertex processing and a reference device.
ps_2_sw	All the features of ps_2_x	Only supported by a reference device.
ps_3_sw	All the features of ps_3_0	Only supported by a reference device.

Some validations are relaxed for software shaders. This is useful for debugging and prototyping purposes. The following validations are relaxed: (all other validations remain the same)

Validation type	Relaxation
Instruction Counts:	This is relaxed for vs_2_sw, vs_3_sw and ps_2_sw, ps_3_sw. Unlimited instructions are allowed.
Float Constant Counts:	This is relaxed for vs_2_sw, vs_3_sw and ps_2_sw, ps_3_sw. Up to 8192 constants are allowed.
Integer Constant Counts:	This is relaxed for vs_2_sw, vs_3_sw and ps_2_sw, ps_3_sw. Up to 2048 constants are allowed.
Boolean Constant Counts:	This is relaxed for vs_2_sw, vs_3_sw and ps_2_sw, ps_3_sw. Up to 2048 constants are allowed.
Dependent-read depth:	This is relaxed for ps_2_sw. Like in vs_3_0 and ps_3_0, unlimited dependent reads are allowed.

Validation type	Relaxation
Number of flow control instructions and labels:	This is relaxed for vs_2_sw. Unlimited flow control instructions and upto 2048 labels are allowed.
Loop count/start/step:	These are relaxed for vs_2_sw, vs_3_sw, ps_2_sw and ps_3_sw. Iteration start and interation step size for rep and loop instructions are 32-bit signed intergers. Interation count can be up to MAX_INT/64.
Read-port limits:	vs_2_sw, vs_3_sw, ps_2_sw and ps_3_sw have no read-port limit.
Number of interpolators:	There are 16 Registers - vs_3_0 (o#) in vs_3_sw and 10 ps_3_0 Registers (v#) for ps_3_sw.

Related topics

[Asm Shader Reference](#)

D3DCompiler Reference

Article • 11/20/2019 • 2 minutes to read

The Direct3D API defines several API elements to compile shader code.

- [Functions](#)
- [Structures](#)
- [Enumerations](#)
- [Constants](#)

Related topics

[Reference for HLSL](#)

Compiler functions (HLSL reference)

Article • 08/19/2021 • 3 minutes to read

This section contains information about the following Direct3D HLSL compiler functions:

In this section

Topic	Description
D3D11Reflect	Gets a pointer to a reflection interface.
D3DCompile	Compile HLSL code or an effect file into bytecode for a given target.
D3DCompile2	Compiles Microsoft High Level Shader Language (HLSL) code into bytecode for a given target.
D3DCompileFromFile	<p>[!Note]</p> <p>You can use this API to develop your Windows Store apps, but you can't use it in apps that you submit to the Windows Store. Refer to the section, "Compiling shaders for UWP", in the remarks for D3DCompile2.</p> <p>Compiles HLSL code into bytecode for a given target.</p>
D3DCompressShaders	<p>[!Note]</p> <p>You can use this API to develop your Windows Store apps, but you can't use it in apps that you submit to the Windows Store.</p> <p>Compresses a set of shaders into a more compact form.</p>
D3DCreateBlob	Creates a buffer.

Topic	Description
D3DCreateFunctionLinkingGraph	<p>Creates a function-linking-graph interface.</p> <p>[!Note] This function is part of the HLSL shader linking technology that you can use on all Direct3D 11 platforms to create precompiled HLSL functions, package them into libraries, and link them into full shaders at run time.</p>
D3DCreateLinker	<p>Creates a linker interface.</p> <p>[!Note] This function is part of the HLSL shader linking technology that you can use on all Direct3D 11 platforms to create precompiled HLSL functions, package them into libraries, and link them into full shaders at run time.</p>
D3DDecompressShaders	<p>[!Note] You can use this API to develop your Windows Store apps, but you can't use it in apps that you submit to the Windows Store.</p> <p>Decompresses one or more shaders from a compressed set.</p>
D3DDisassemble	Disassembles compiled HLSL code.
D3DDisassemble10Effect	Disassembles compiled HLSL code from a Direct3D10 effect.
D3DDisassemble11Trace	Disassembles a section of compiled HLSL code that is specified by shader trace steps.
D3DDisassembleRegion	Disassembles a specific region of compiled HLSL code.
D3DGetBlobPart	Retrieves a specific part from a compilation result.

Topic	Description
D3DGetDebugInfo	<p>[!Note]</p> <p>You can use this API to develop your Windows Store apps, but you can't use it in apps that you submit to the Windows Store.</p> <p>Gets shader debug information.</p>
D3DGetInputAndOutputSignatureBlob	<p>[!Note]</p> <p>D3DGetInputAndOutputSignatureBlob may be altered or unavailable for releases after Windows 8.1. Instead use D3DGetBlobPart with the D3D_BLOB_INPUT_AND_OUTPUT_SIGNATURE_BLOB value.</p> <p>Gets the input and output signatures from a compilation result.</p>
D3DGetInputSignatureBlob	<p>[!Note]</p> <p>D3DGetInputSignatureBlob may be altered or unavailable for releases after Windows 8.1. Instead use D3DGetBlobPart with the D3D_BLOB_INPUT_SIGNATURE_BLOB value.</p> <p>Gets the input signature from a compilation result.</p>
D3DGetOutputSignatureBlob	<p>[!Note]</p> <p>D3DGetOutputSignatureBlob may be altered or unavailable for releases after Windows 8.1. Instead use D3DGetBlobPart with the D3D_BLOB_OUTPUT_SIGNATURE_BLOB value.</p> <p>Gets the output signature from a compilation result.</p>
D3DGetTraceInstructionOffsets	Retrieves the byte offsets for instructions within a section of shader code.

Topic	Description
D3DLoadModule	<p>Creates a shader module interface from source data for the shader module.</p> <p>[!Note] This function is part of the HLSL shader linking technology that you can use on all Direct3D 11 platforms to create precompiled HLSL functions, package them into libraries, and link them into full shaders at run time.</p>
D3DPreprocess	Preprocesses uncompiled HLSL code.
D3DReadFileToBlob	<p>[!Note] You can use this API to develop your Windows Store apps, but you can't use it in apps that you submit to the Windows Store.</p> <p>Reads a file that is on disk into memory.</p>
D3DReflect	Gets a pointer to a reflection interface.
D3DReflectLibrary	<p>Creates a library-reflection interface from source data that contains an HLSL library of functions.</p> <p>[!Note] This function is part of the HLSL shader linking technology that you can use on all Direct3D 11 platforms to create precompiled HLSL functions, package them into libraries, and link them into full shaders at run time.</p>
D3DSetBlobPart	Sets information in a compilation result.
D3DStripShader	Removes unwanted blobs from a compilation result.

Topic	Description
D3DWriteBlobToFile	<p>[!Note] You can use this API to develop your Windows Store apps, but you can't use it in apps that you submit to the Windows Store.</p> <p>Writes a memory blob to a file on disk.</p>

Related topics

[D3DCompiler Reference](#)

D3D11Reflect function

Article • 03/15/2021 • 2 minutes to read

Gets a pointer to a reflection interface.

Syntax

syntax

```
HRESULT D3D11Reflect(  
    in  LPVOID pSrcData,  
    in  SIZE_T SrcDataSize,  
    out ID3D11ShaderReflection ppReflector  
) ;
```

Parameters

pSrcData [in]

Type: [LPCVOID](#)

A pointer to source data as compiled HLSL code.

SrcDataSize [in]

Type: [SIZE_T](#)

Length of *pSrcData*.

ppReflector [out]

Type: [ID3D11ShaderReflection**](#)

The address of a pointer to the [ID3D11ShaderReflection](#) interface.

Return value

Type: [HRESULT](#)

Returns one of the return codes described in the topic [Direct3D 11 Return Codes](#).

Remarks

The inline **D3D11Reflect** compiler function is a wrapper for the **D3DReflect** compiler function. **D3D11Reflect** can retrieve only a **ID3D11ShaderReflection** interface from a shader. **D3DReflect** can retrieve a **ID3D11ShaderReflection** interface or a Direct3D 10 or Direct3D 10.1 reflection interface, for example, **ID3D10ShaderReflection**.

Shader code contains metadata that can be inspected using the reflection APIs.

The following code shows how to retrieve a **ID3D11ShaderReflection** interface from a shader.

C++

```
pd3dDevice->CreatePixelShader( pPixelShaderBuffer->GetBufferPointer(),
                                pPixelShaderBuffer->GetBufferSize(),
                                g_pPSClassLinkage, &g_pPixelShader );

ID3D11ShaderReflection* pReflector = NULL;
D3D11Reflect( pPixelShaderBuffer->GetBufferPointer(), pPixelShaderBuffer-
               >GetBufferSize(),
               &pReflector);
```

Requirements

Requirement	Value
Header	D3DCompiler.inl
Library	D3dcompiler_47.lib
DLL	D3dcompiler_47.dll

See also

[Functions](#)

D3DCompile function (d3dcompiler.h)

Article • 12/02/2012 minutes to read

Compile HLSL code or an effect file into bytecode for a given target.

Syntax

C++

```
HRESULT D3DCompile(
    [in]           LPCVOID             pSrcData,
    [in]           SIZE_T              SrcDataSize,
    [in, optional] LPCSTR              pSourceName,
    [in, optional] const D3D_SHADER_MACRO *pDefines,
    [in, optional] ID3DInclude        *pInclude,
    [in, optional] LPCSTR              pEntryPoint,
    [in]           LPCSTR              pTarget,
    [in]           UINT                Flags1,
    [in]           UINT                Flags2,
    [out]          ID3DBlob            **ppCode,
    [out, optional] ID3DBlob           **ppErrorMsgs
);
```

Parameters

[in] `pSrcData`

Type: [LPCVOID](#)

A pointer to uncompiled shader data; either ASCII HLSL code or a compiled effect.

[in] `SrcDataSize`

Type: [SIZE_T](#)

Length of `pSrcData`.

[in, optional] `pSourceName`

Type: [LPCSTR](#)

You can use this parameter for strings that specify error messages. If not used, set to `NULL`.

[in, optional] `pDefines`

Type: [const D3D_SHADER_MACRO*](#)

An optional array of [D3D_SHADER_MACRO](#) structures that define shader macros. Each macro definition contains a name and a null-terminated definition. If not used, set to **NULL**. The last structure in the array serves as a terminator and must have all members set to **NULL**.

[in, optional] pInclude

Type: [ID3DInclude*](#)

Optional. A pointer to an [ID3DInclude](#) for handling include files. Setting this to **NULL** will cause a compile error if a shader contains a #include. You can pass the **D3D_COMPILE_STANDARD_FILE_INCLUDE** macro, which is a pointer to a default include handler. This default include handler includes files that are relative to the current directory and files that are relative to the directory of the initial source file. When you use **D3D_COMPILE_STANDARD_FILE_INCLUDE**, you must specify the source file name in the *pSourceName* parameter; the compiler will derive the initial relative directory from *pSourceName*.

C++

```
#define D3D_COMPILE_STANDARD_FILE_INCLUDE ((ID3DInclude*) (UINT_PTR)1)
```

[in, optional] pEntryPoint

Type: [LPCSTR](#)

The name of the shader entry point function where shader execution begins. When you compile using a fx profile (for example, fx_4_0, fx_5_0, and so on), **D3DCompile** ignores *pEntryPoint*. In this case, we recommend that you set *pEntryPoint* to **NULL** because it is good programming practice to set a pointer parameter to **NULL** if the called function will not use it. For all other shader profiles, a valid *pEntryPoint* is required.

[in] pTarget

Type: [LPCSTR](#)

A string that specifies the shader target or set of shader features to compile against. The shader target can be shader model 2, shader model 3, shader model 4, or shader model 5. The target can also be an effect type (for example, fx_4_1). For info about the targets that various profiles support, see [Specifying Compiler Targets](#).

[in] Flags1

Type: [UINT](#)

Flags defined by [D3D compile constants](#).

[in] Flags2

Type: [UINT](#)

Flags defined by [D3D compile effect constants](#). When you compile a shader and not an effect file, [D3DCompile](#) 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.

[out] ppCode

Type: [ID3DBlob**](#)

A pointer to a variable that receives a pointer to the [ID3DBlob](#) interface that you can use to access the compiled code.

[out, optional] ppErrorMsgs

Type: [ID3DBlob**](#)

A pointer to a variable that receives a pointer to the [ID3DBlob](#) interface that you can use to access compiler error messages, or [NULL](#) if there are no errors.

Return value

Type: [HRESULT](#)

Returns one of the [Direct3D 11 return codes](#).

Remarks

The difference between [D3DCompile](#) and [D3DCompile2](#) is that the latter method takes some optional parameters that can be used to control some aspects of how bytecode is generated. If this extra flexibility is not required, there is no performance gain from using [D3DCompile2](#).

Requirements

Target Platform	Windows
Header	d3dcompiler.h
Library	d3dcompiler.lib
DLL	d3dcompiler_47.dll

See also

[Functions](#)

D3DCompile2 function (d3dcompiler.h)

Article • 07/27/2022 5 minutes to read

Compiles Microsoft High Level Shader Language (HLSL) code into bytecode for a given target.

Syntax

C++

```
HRESULT D3DCompile2(
    [in]           LPCVOID          pSrcData,
    [in]           SIZE_T           SrcDataSize,
    [in, optional] LPCSTR           pSourceName,
    [in, optional] const D3D_SHADER_MACRO *pDefines,
    [in, optional] ID3DInclude      *pInclude,
    [in]           LPCSTR           pEntrypoint,
    [in]           LPCSTR           pTarget,
    [in]           UINT             Flags1,
    [in]           UINT             Flags2,
    [in]           UINT             SecondaryDataFlags,
    [in, optional] LPCVOID          pSecondaryData,
    [in]           SIZE_T           SecondaryDataSize,
    [out]          ID3DBlob        **ppCode,
    [out, optional] ID3DBlob        **ppErrorMsgs
);
```

Parameters

[in] *pSrcData*

Type: [LPCVOID](#)

A pointer to uncompiled shader data (ASCII HLSL code).

[in] *SrcDataSize*

Type: [SIZE_T](#)

The size, in bytes, of the block of memory that *pSrcData* points to.

[in, optional] *pSourceName*

Type: [LPCSTR](#)

An optional pointer to a constant null-terminated string containing the name that identifies the source data to use in error messages. If not used, set to **NULL**.

[in, optional] pDefines

Type: [const D3D_SHADER_MACRO*](#)

An optional array of [D3D_SHADER_MACRO](#) structures that define shader macros. Each macro definition contains a name and a null-terminated definition. If not used, set to **NULL**. The last structure in the array serves as a terminator and must have all members set to **NULL**.

[in, optional] pInclude

Type: [ID3DInclude*](#)

A pointer to an [ID3DInclude](#) interface that the compiler uses to handle include files. If you set this parameter to **NULL** and the shader contains a #include, a compile error occurs. You can pass the [D3D_COMPILE_STANDARD_FILE_INCLUDE](#) macro, which is a pointer to a default include handler. This default include handler includes files that are relative to the current directory and files that are relative to the directory of the initial source file. When you use [D3D_COMPILE_STANDARD_FILE_INCLUDE](#), you must specify the source file name in the *pSourceName* parameter; the compiler will derive the initial relative directory from *pSourceName*.

```
#define D3D_COMPILE_STANDARD_FILE_INCLUDE ((ID3DInclude*) (UINT_PTR)1)
```

[in] pEntrypoint

Type: [LPCSTR](#)

A pointer to a constant null-terminated string that contains the name of the shader entry point function where shader execution begins. When you compile an effect, [D3DCompile2](#) ignores *pEntrypoint*; we recommend that you set *pEntrypoint* to **NULL** because it is good programming practice to set a pointer parameter to **NULL** if the called function will not use it.

[in] pTarget

Type: [LPCSTR](#)

A pointer to a constant null-terminated string that specifies the shader target or set of shader features to compile against. The shader target can be a shader model (for

example, shader model 2, shader model 3, shader model 4, or shader model 5). The target can also be an effect type (for example, fx_4_1). For info about the targets that various profiles support, see [Specifying Compiler Targets](#).

[in] Flags1

Type: [UINT](#)

A combination of shader [D3D compile constants](#) that are combined by using a bitwise **OR** operation. The resulting value specifies how the compiler compiles the HLSL code.

[in] Flags2

Type: [UINT](#)

A combination of effect [D3D compile effect constants](#) that are combined by using a bitwise **OR** operation. The resulting value specifies how the compiler compiles the effect. When you compile a shader and not an effect file, [D3DCompile2](#) 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.

[in] SecondaryDataFlags

Type: [UINT](#)

A combination of the following flags that are combined by using a bitwise **OR** operation. The resulting value specifies how the compiler compiles the HLSL code.

Flag	Description
D3DCOMPILE_SECDATA_MERGE_UAV_SLOTS (0x01)	Merge unordered access view (UAV) slots in the secondary data that the <i>pSecondaryData</i> parameter points to.
D3DCOMPILE_SECDATA_PRESERVE_TEMPLATE_SLOTS (0x02)	Preserve template slots in the secondary data that the <i>pSecondaryData</i> parameter points to.
D3DCOMPILE_SECDATA_REQUIRE_TEMPLATE_MATCH (0x04)	Require that templates in the secondary data that the <i>pSecondaryData</i> parameter points to match when the compiler compiles the HLSL code.

If *pSecondaryData* is **NULL**, set to zero.

[in, optional] pSecondaryData

Type: [LPCVOID](#)

A pointer to secondary data. If you don't pass secondary data, set to **NULL**. Use this secondary data to align UAV slots in two shaders. Suppose shader A has UAVs and they are bound to some slots. To compile shader B such that UAVs with the same names are mapped in B to the same slots as in A, pass A's byte code to [D3DCompile2](#) as the secondary data.

[in] `SecondaryDataSize`

Type: [SIZE_T](#)

The size, in bytes, of the block of memory that *pSecondaryData* points to. If *pSecondaryData* is **NULL**, set to zero.

[out] `ppCode`

Type: [ID3DBlob**](#)

A pointer to a variable that receives a pointer to the [ID3DBlob](#) interface that you can use to access the compiled code.

[out, optional] `ppErrorMsgs`

Type: [ID3DBlob**](#)

A pointer to a variable that receives a pointer to the [ID3DBlob](#) interface that you can use to access compiler error messages, or **NULL** if there are no errors.

Return value

Type: [HRESULT](#)

Returns one of the [Direct3D 11 return codes](#).

Remarks

The difference between [D3DCompile2](#) and [D3DCompile](#) is that [D3DCompile2](#) takes some optional parameters (*SecondaryDataFlags*, *pSecondaryData* and *SecondaryDataSize*) that can be used to control some aspects of how bytecode is generated. Refer to the descriptions of these parameters for more details. There is no difference otherwise to the efficiency of the bytecode generated between [D3DCompile2](#) and [D3DCompile](#).

Compiling shaders for UWP

To compile offline shaders the recommended approach is to use the [Effect-compiler tool](#). If you can't compile all of your shaders ahead of time, then consider compiling the more expensive ones and the ones that your startup and most performance-sensitive paths require, and compiling the rest at runtime. You can use a process similar to the following to compile a loaded or generated shader in a UWP application without blocking your user interface thread.

- Using Visual Studio 2015+ to develop the UWP app, add the new item "shader.hlsl".
 - In the **Solution Folder** view of Visual Studio, select the **shaders.hlsl** item, right-click for **Properties**.
 - Make sure the item **Content** is set to **Yes**.
 - Make sure the **Item Type** is set to **Text**.
 - Add a button to XAML, name it appropriately ("TheButton" in this example), and add a **Click** handler.
- Now add these includes to your .cpp file:

syntax

```
#include <ppltasks.h>
#include <d3dcompiler.h>
#include <Robuffer.h>
```

- Use the following code to call **D3DCompile2**. Note that there's no error checking or handling here, and also that this code demonstrates that you can do both I/O and compilation in the background, which leaves your UI more responsive.

C++/CX

```
void App1::DirectXPage::TheButton_Click(Platform::Object^ sender,
Windows::UI::Xaml::RoutedEventArgs^ e)
{
    std::shared_ptr<Microsoft::WRL::ComPtr<ID3DBlob>> blobRef =
    std::make_shared<Microsoft::WRL::ComPtr<ID3DBlob>>();

    // Load a file and compile it.
    auto fileOp = Windows::ApplicationModel::Package::Current-
>InstalledLocation->GetFileAsync(L"shader.hlsl");
    create_task(fileOp).then([this](Windows::Storage::StorageFile^ file) ->
IAsyncOperation<Windows::Storage::Streams::IBuffer^>^
    {
        // Do file I/O in background thread (use_arbitrary).
        return Windows::Storage::FileIO::ReadBufferAsync(file);
    }, task_continuation_context::use_arbitrary());
}
```

```

    .then([this, blobRef](Windows::Storage::Streams::IBuffer^ buffer)
{
    // Do compilation in background thread (use_arbitrary).

    // Cast to Object^, then to its underlying IInspectable interface.
    Microsoft::WRL::ComPtr<IInspectable>
insp(reinterpret_cast<IInspectable*>(buffer));

    // Query the IBufferByteAccess interface.
    Microsoft::WRL::ComPtr<Windows::Storage::Streams::IBufferByteAccess>
bufferByteAccess;
    insp.As(&bufferByteAccess);

    // Retrieve the buffer data.
    byte *pBytes = nullptr;
    bufferByteAccess->Buffer(&pBytes);

    Microsoft::WRL::ComPtr<ID3DBlob> blob;
    Microsoft::WRL::ComPtr<ID3DBlob> errMsgs;
    D3DCompile2(pBytes, buffer->Length, "shader.hlsl", nullptr, nullptr,
"main", "ps_5_0", 0, 0, 0, nullptr, 0, blob.GetAddressOf(),
errMsgs.GetAddressOf());
    *blobRef = blob;
}, task_continuation_context::use_arbitrary())
.then([this, blobRef]()
{
    // Update UI / use shader on foreground thread.
    wchar_t message[40];
    swprintf_s(message, L"blob is %u bytes long", (unsigned)(*blobRef)-
>GetBufferSize());
    this->TheButton->Content = ref new Platform::String(message);
}, task_continuation_context::use_current());
}

```

Requirements

Target Platform	Windows
Header	d3dcompiler.h
Library	D3DCompiler.lib
DLL	D3DCompiler_47.dll

See also

- [Functions](#)

- Specifying D3D12 Root Signatures in HLSL

D3DCompileFromFile function (d3dcompiler.h)

Article07/27/2022

Note You can use this API to develop your Windows Store apps, but you can't use it in apps that you submit to the Windows Store. Refer to the section, "Compiling shaders for UWP", in the remarks for [D3DCompile2](#).

Compiles Microsoft High Level Shader Language (HLSL) code into bytecode for a given target.

Syntax

C++

```
HRESULT D3DCompileFromFile(
    [in]          LPCWSTR             pFileName,
    [in, optional] const D3D_SHADER_MACRO *pDefines,
    [in, optional] ID3DInclude        *pInclude,
    [in]          LPCSTR              pEntryPoint,
    [in]          LPCSTR              pTarget,
    [in]          UINT                Flags1,
    [in]          UINT                Flags2,
    [out]         ID3DBlob           **ppCode,
    [out, optional] ID3DBlob           **ppErrorMsgs
);
```

Parameters

[in] pFileName

A pointer to a constant null-terminated string that contains the name of the file that contains the shader code.

[in, optional] pDefines

An optional array of [D3D_SHADER_MACRO](#) structures that define shader macros. Each macro definition contains a name and a null-terminated definition. If not used, set to **NULL**. The last structure in the array serves as a terminator and must have all members set to **NULL**.

[in, optional] *pInclude*

An optional pointer to an [ID3DInclude](#) interface that the compiler uses to handle include files. If you set this parameter to **NULL** and the shader contains a #include, a compile error occurs. You can pass the **D3D_COMPILE_STANDARD_FILE_INCLUDE** macro, which is a pointer to a default include handler. This default include handler includes files that are relative to the current directory.

```
#define D3D_COMPILE_STANDARD_FILE_INCLUDE ((ID3DInclude*)(UINT_PTR)1)
```

[in] *pEntryPoint*

A pointer to a constant null-terminated string that contains the name of the shader entry point function where shader execution begins. When you compile an effect, **D3DCompileFromFile** ignores *pEntryPoint*; we recommend that you set *pEntryPoint* to **NULL** because it is good programming practice to set a pointer parameter to **NULL** if the called function will not use it.

[in] *pTarget*

A pointer to a constant null-terminated string that specifies the shader target or set of shader features to compile against. The shader target can be a shader model (for example, shader model 2, shader model 3, shader model 4, or shader model 5 and later). The target can also be an effect type (for example, fx_4_1). For info about the targets that various profiles support, see [Specifying Compiler Targets](#).

[in] *Flags1*

A combination of shader [compile options](#) that are combined by using a bitwise **OR** operation. The resulting value specifies how the compiler compiles the HLSL code.

[in] *Flags2*

A combination of effect [compile options](#) that are combined by using a bitwise **OR** operation. The resulting value specifies how the compiler compiles the effect. When you compile a shader and not an effect file, **D3DCompileFromFile** 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.

[out] *ppCode*

A pointer to a variable that receives a pointer to the [ID3DBlob](#) interface that you can use to access the compiled code.

[out, optional] ppErrorMsgs

An optional pointer to a variable that receives a pointer to the [ID3DBlob](#) interface that you can use to access compiler error messages, or **NULL** if there are no errors.

Return value

Returns one of the [Direct3D 11 return codes](#).

Remarks

Note The D3dcompiler_44.dll or later version of the file contains the **D3DCompileFromFile** compiler function.

Requirements

Target Platform	Windows
Header	d3dcompiler.h
Library	D3DCompiler.lib
DLL	D3DCompiler_47.dll

See also

[Functions](#)

D3DCompressShaders function (d3dcompiler.h)

Article 10/13/2021

Note You can use this API to develop your Windows Store apps, but you can't use it in apps that you submit to the Windows Store.

Compresses a set of shaders into a more compact form.

Syntax

C++

```
HRESULT D3DCompressShaders(
    [in]    UINT          uNumShaders,
    [in]    D3D_SHADER_DATA *pShaderData,
    [in]    UINT          uFlags,
    [out]   ID3DBlob     **ppCompressedData
);
```

Parameters

[in] uNumShaders

Type: [UINT](#)

The number of shaders to compress.

[in] pShaderData

Type: [D3D_SHADER_DATA*](#)

An array of [D3D_SHADER_DATA](#) structures that describe the set of shaders to compress.

[in] uFlags

Type: [UINT](#)

Flags that indicate how to compress the shaders. Currently, only the D3D_COMPRESS_SHADER_KEEP_ALL_PARTS (0x00000001) flag is defined.

[out] ppCompressedData

Type: [ID3DBlob**](#)

The address of a pointer to the [ID3DBlob](#) interface that is used to retrieve the compressed shader data.

Return value

Type: [HRESULT](#)

Returns one of the [Direct3D 11 return codes](#).

Requirements

Target Platform	Windows
Header	d3dcompiler.h
Library	D3DCompiler.lib
DLL	D3DCompiler_47.dll

See also

[Functions](#)

D3DCreateBlob function (d3dcompiler.h)

Article • 10/13/2021 2 minutes to read

Creates a buffer.

Syntax

C++

```
HRESULT D3DCreateBlob(
    [in] SIZE_T Size,
    [out] ID3DBlob **ppBlob
);
```

Parameters

[in] `Size`

Type: [SIZE_T](#)

Number of bytes in the blob.

[out] `ppBlob`

Type: [ID3DBlob**](#)

The address of a pointer to the [ID3DBlob](#) interface that is used to retrieve the buffer.

Return value

Type: [HRESULT](#)

Returns one of the [Direct3D 11 return codes](#).

Remarks

The latest D3dcompiler_nn.dll contains the [D3DCreateBlob](#) compiler function. Therefore, you are no longer required to create and use an arbitrary length data buffer by using the [D3D10CreateBlob](#) function that is contained in D3d10.dll.

Requirements

Target Platform	Windows
Header	d3dcompiler.h
Library	D3DCompiler.lib
DLL	D3DCompiler_47.dll

See also

[Functions](#)

D3DCreateFunctionLinkingGraph function (d3dcompiler.h)

Article • 10/13/2021 2 minutes to read

Creates a function-linking-graph interface.

Note This function is part of the HLSL shader linking technology that you can use on all Direct3D 11 platforms to create precompiled HLSL functions, package them into libraries, and link them into full shaders at run time.

Syntax

C++

```
HRESULT D3DCreateFunctionLinkingGraph(
    [in]  UINT           uFlags,
    [out] ID3D11FunctionLinkingGraph **ppFunctionLinkingGraph
);
```

Parameters

[in] `uFlags`

Type: [UINT](#)

Reserved

[out] `ppFunctionLinkingGraph`

Type: [ID3D11FunctionLinkingGraph**](#)

A pointer to a variable that receives a pointer to the [ID3D11FunctionLinkingGraph](#) interface that is used for constructing shaders that consist of a sequence of precompiled function calls.

Return value

Type: [HRESULT](#)

Returns S_OK if successful; otherwise, returns one of the [Direct3D 11 Return Codes](#).

Remarks

Note The D3dcompiler_47.dll or later version of the DLL contains the [D3DCreateFunctionLinkingGraph](#) function.

Requirements

Target Platform	Windows
Header	d3dcompiler.h
Library	D3DCompiler.lib
DLL	D3DCompiler_47.dll

See also

[Functions](#)

[ID3D11FunctionLinkingGraph](#)

D3DCreateLinker function (d3dcompiler.h)

Article 10/13/2021

Creates a linker interface.

Note This function is part of the HLSL shader linking technology that you can use on all Direct3D 11 platforms to create precompiled HLSL functions, package them into libraries, and link them into full shaders at run time.

Syntax

C++

```
HRESULT D3DCreateLinker(
    [out] ID3D11Linker **ppLinker
);
```

Parameters

[out] ppLinker

Type: [ID3D11Linker**](#)

A pointer to a variable that receives a pointer to the [ID3D11Linker](#) interface that is used to link a shader module.

Return value

Type: [HRESULT](#)

Returns S_OK if successful; otherwise, returns one of the [Direct3D 11 Return Codes](#).

Remarks

Note The D3dcompiler_47.dll or later version of the DLL contains the D3DCreateLinker function.

Requirements

Target Platform	Windows
Header	d3dcompiler.h
Library	D3DCompiler.lib
DLL	D3DCompiler_47.dll

See also

[Functions](#)

[ID3D11Linker](#)

D3DDecompressShaders function (d3dcompiler.h)

Article • 10/13/2021 2 minutes to read

Note You can use this API to develop your Windows Store apps, but you can't use it in apps that you submit to the Windows Store.

Decompresses one or more shaders from a compressed set.

Syntax

C++

```
HRESULT D3DDecompressShaders(
    [in]          LPCVOID  pSrcData,
    [in]          SIZE_T   SrcDataSize,
    [in]          UINT     uNumShaders,
    [in]          UINT     ustartIndex,
    [in, optional] UINT     *pIndices,
    [in]          UINT     uFlags,
    [out]         ID3DBlob **ppShaders,
    [out, optional] UINT     *pTotalShaders
);
```

Parameters

[in] *pSrcData*

Type: [LPCVOID](#)

A pointer to uncompiled shader data; either ASCII HLSL code or a compiled effect.

[in] *SrcDataSize*

Type: [SIZE_T](#)

Length of uncompiled shader data that *pSrcData* points to.

[in] *uNumShaders*

Type: [UINT](#)

The number of shaders to decompress.

[in] `uStartIndex`

Type: [UINT](#)

The index of the first shader to decompress.

[in, optional] `pIndices`

Type: [UINT*](#)

An array of indexes that represent the shaders to decompress.

[in] `uFlags`

Type: [UINT](#)

Flags that indicate how to decompress. Currently, no flags are defined.

[out] `ppShaders`

Type: [ID3DBlob**](#)

The address of a pointer to the [ID3DBlob](#) interface that is used to retrieve the decompressed shader data.

[out, optional] `pTotalShaders`

Type: [UINT*](#)

A pointer to a variable that receives the total number of shaders that `D3DDecompressShaders` decompressed.

Return value

Type: [HRESULT](#)

Returns one of the [Direct3D 11 return codes](#).

Requirements

Target Platform	Windows

Header	d3dcompiler.h
Library	D3DCompiler.lib
DLL	D3DCompiler_47.dll

See also

[Functions](#)

D3DDisassemble function (d3dcompiler.h)

Article 10/13/2021

Disassembles compiled HLSL code.

Syntax

C++

```
HRESULT D3DDisassemble(
    [in]          LPCVOID pSrcData,
    [in]          SIZE_T  SrcDataSize,
    [in]          UINT    Flags,
    [in, optional] LPCSTR  szComments,
    [out]         ID3DBlob **ppDisassembly
);
```

Parameters

[in] *pSrcData*

Type: [LPCVOID](#)

A pointer to source data as compiled HLSL code.

[in] *SrcDataSize*

Type: [SIZE_T](#)

Length of *pSrcData*.

[in] *Flags*

Type: [UINT](#)

Flags affecting the behavior of **D3DDisassemble**. *Flags* can be a combination of zero or more of the following values.

Flag	Description
<code>D3D_DISASM_ENABLE_COLOR_CODE</code>	Enable the output of color codes.

D3D_DISASM_ENABLE_DEFAULT_VALUE_PRINTS	Enable the output of default values.
D3D_DISASM_ENABLE_INSTRUCTION_NUMBERING	Enable instruction numbering.
D3D_DISASM_ENABLE_INSTRUCTION_CYCLE	No effect.
D3D_DISASM_DISABLE_DEBUG_INFO	Disable debug information.
D3D_DISASM_ENABLE_INSTRUCTION_OFFSET	Enable instruction offsets.
D3D_DISASM_INSTRUCTION_ONLY	Disassemble instructions only.
D3D_DISASM_PRINT_HEX_LITERALS	Use hex symbols in disassemblies.

[in, optional] szComments

Type: [LPCSTR](#)

The comment string at the top of the shader that identifies the shader constants and variables.

[out] ppDisassembly

Type: [ID3DBlob**](#)

A pointer to a buffer that receives the [ID3DBlob](#) interface that accesses assembly text.

Return value

Type: [HRESULT](#)

Returns one of the [Direct3D 11 return codes](#).

Requirements

Target Platform	Windows
Header	d3dcompiler.h
Library	D3dcompiler_47.lib
DLL	D3dcompiler_47.dll

See also

Functions

D3DDisassemble10Effect function (d3dcompiler.h)

Article • 10/13/2021 2 minutes to read

Disassembles compiled HLSL code from a Direct3D10 effect.

Syntax

C++

```
HRESULT D3DDisassemble10Effect(
    [in] ID3D10Effect *pEffect,
    [in] UINT          Flags,
    [out] ID3DBlob    **ppDisassembly
);
```

Parameters

[in] pEffect

Type: [ID3D10Effect*](#)

A pointer to source data as compiled HLSL code.

[in] Flags

Type: [UINT](#)

Shader [compile options](#).

[out] ppDisassembly

Type: [ID3DBlob**](#)

A pointer to a buffer that receives the [ID3DBlob](#) interface that contains disassembly text.

Return value

Type: [HRESULT](#)

Returns one of the [Direct3D 11 return codes](#).

Requirements

Target Platform	Windows
Header	d3dcompiler.h
Library	D3dcompiler_47.lib
DLL	D3dcompiler_47.dll

See also

[Functions](#)

D3DDisassemble11Trace function (d3d11shadertracing.h)

Article • 10/13/2021 2 minutes to read

Disassembles a section of compiled Microsoft High Level Shader Language (HLSL) code that is specified by shader trace steps.

Syntax

C++

```
HRESULT D3DDisassemble11Trace(
    [in]    LPCVOID           pSrcData,
    [in]    SIZE_T            SrcDataSize,
    [in]    ID3D11ShaderTrace *pTrace,
    [in]    UINT               StartStep,
    [in]    UINT               NumSteps,
    [in]    UINT               Flags,
    [out]   ID3D10Blob        **ppDisassembly
);
```

Parameters

[in] pSrcData

Type: **LPCVOID**

A pointer to compiled shader data.

[in] SrcDataSize

Type: **SIZE_T**

The size, in bytes, of the block of memory that pSrcData points to.

[in] pTrace

Type: **ID3D11ShaderTrace***

A pointer to the ID3D11ShaderTrace interface for the shader trace information object.

[in] StartStep

Type: **UINT**

The number of the step in the trace from which D3DDisassemble11Trace starts the disassembly.

[in] **NumSteps**

Type: **UINT**

The number of trace steps to disassemble.

[in] **Flags**

Type: **UINT**

A combination of zero or more of the following flags that are combined by using a bitwise OR operation. The resulting value specifies how D3DDisassemble11Trace disassembles the compiled shader data.

Flag	Description
D3D_DISASM_ENABLE_COLOR_CODE (0x01)	Enable the output of color codes.
D3D_DISASM_ENABLE_DEFAULT_VALUE_PRINTS (0x02)	Enable the output of default values.
D3D_DISASM_ENABLE_INSTRUCTION_NUMBERING (0x04)	Enable instruction numbering.
D3D_DISASM_ENABLE_INSTRUCTION_CYCLE (0x08)	No effect.
D3D_DISASM_DISABLE_DEBUG_INFO (0x10)	Disable the output of debug information.
D3D_DISASM_ENABLE_INSTRUCTION_OFFSET (0x20)	Enable the output of instruction offsets.
D3D_DISASM_INSTRUCTION_ONLY (0x40)	Enable the output of the instruction cycle per step in D3DDisassemble11Trace. This flag is similar to the D3D_DISASM_ENABLE_INSTRUCTION_NUMBERING and D3D_DISASM_ENABLE_INSTRUCTION_OFFSET flags. This flag has no effect in the D3DDisassembleRegion function. Cycle information comes from the trace; therefore, cycle information is available only in the trace disassembly.

[out] **ppDisassembly**

Type: **ID3D10Blob****

A pointer to a buffer that receives the ID3DBlob interface that accesses the disassembled HLSL code.

Return value

Type: **HRESULT**

This method returns an HRESULT error code.

Remarks

D3DDisassemble11Trace walks the steps of a shader trace and outputs appropriate disassembly for each step that is based on the step's instruction index. The disassembly is annotated with register-value information from the trace. The behavior of D3DDisassemble11Trace differs from D3DDisassemble in that instead of the static disassembly of a compiled shader that D3DDisassemble performs, D3DDisassemble11Trace provides an execution trace that is based on the shader trace information.

Requirements

Minimum supported client	Windows 8 [desktop apps UWP apps]
Minimum supported server	Windows Server 2012 [desktop apps UWP apps]
Target Platform	Windows
Header	d3d11shadertracing.h
DLL	D3D11SDKLayers.dll; D3D11_1SDKLayers.dll; D3D11_2SDKLayers.dll

See also

[Shader Functions](#)

D3DDisassembleRegion function (d3dcompiler.h)

Article 10/13/2021

Disassembles a specific region of compiled Microsoft High Level Shader Language (HLSL) code.

Syntax

C++

```
HRESULT D3DDisassembleRegion(
    [in]          LPCVOID  pSrcData,
    [in]          SIZE_T   SrcDataSize,
    [in]          UINT     Flags,
    [in, optional] LPCSTR   szComments,
    [in]          SIZE_T   StartByteOffset,
    [in]          SIZE_T   NumInsts,
    [out, optional] SIZE_T   *pFinishByteOffset,
    [out]         ID3DBlob **ppDisassembly
);
```

Parameters

[in] `pSrcData`

A pointer to compiled shader data.

[in] `SrcDataSize`

The size, in bytes, of the block of memory that `pSrcData` points to.

[in] `Flags`

A combination of zero or more of the following flags that are combined by using a bitwise OR operation. The resulting value specifies how `D3DDisassembleRegion` disassembles the compiled shader data.

Flag	Description
<code>D3D_DISASM_ENABLE_COLOR_CODE</code> (0x01)	Enable the output of color codes.
<code>D3D_DISASM_ENABLE_DEFAULT_VALUE_PRINTS</code>	Enable the output of default values.

(0x02)	
D3D_DISASM_ENABLE_INSTRUCTION_NUMBERING (0x04)	Enable instruction numbering.
D3D_DISASM_ENABLE_INSTRUCTION_CYCLE (0x08)	No effect.
D3D_DISASM_DISABLE_DEBUG_INFO (0x10)	Disable the output of debug information.
D3D_DISASM_ENABLE_INSTRUCTION_OFFSET (0x20)	Enable the output of instruction offsets.
D3D_DISASM_INSTRUCTION_ONLY (0x40)	This flag has no effect in D3DDisassembleRegion . Cycle information comes from the trace; therefore, cycle information is available only in D3DDisassemble11Trace 's trace disassembly.

[in, optional] szComments

A pointer to a constant null-terminated string at the top of the shader that identifies the shader constants and variables.

[in] StartByteOffset

The number of bytes offset into the compiled shader data where **D3DDisassembleRegion** starts the disassembly.

[in] NumInsts

The number of instructions to disassemble.

[out, optional] pFinishByteOffset

A pointer to a variable that receives the number of bytes offset into the compiled shader data where **D3DDisassembleRegion** finishes the disassembly.

[out] ppDisassembly

A pointer to a buffer that receives the [ID3DBlob](#) interface that accesses the disassembled HLSL code.

Return value

Returns one of the [Direct3D 11 return codes](#).

Remarks

Note The D3dcompiler_44.dll or later version of the file contains the **D3DDisassembleRegion** compiler function.

Requirements

Target Platform	Windows
Header	d3dcompiler.h
Library	D3DCompiler.lib
DLL	D3DCompiler_47.dll

See also

[Functions](#)

D3DGetBlobPart function (d3dcompiler.h)

Article • 10/13/2021 2 minutes to read

Retrieves a specific part from a compilation result.

Syntax

C++

```
HRESULT D3DGetBlobPart(
    [in]    LPCVOID      pSrcData,
    [in]    SIZE_T       SrcDataSize,
    [in]    D3D_BLOB_PART Part,
    [in]    UINT          Flags,
    [out]   ID3DBlob    **ppPart
);
```

Parameters

[in] *pSrcData*

Type: [LPCVOID](#)

A pointer to uncompiled shader data; either ASCII HLSL code or a compiled effect.

[in] *SrcDataSize*

Type: [SIZE_T](#)

Length of uncompiled shader data that *pSrcData* points to.

[in] *Part*

Type: [D3D_BLOB_PART](#)

A [D3D_BLOB_PART](#)-typed value that specifies the part of the buffer to retrieve.

[in] *Flags*

Type: [UINT](#)

Flags that indicate how to retrieve the blob part. Currently, no flags are defined.

[out] ppPart

Type: [ID3DBlob**](#)

The address of a pointer to the [ID3DBlob](#) interface that is used to retrieve the specified part of the buffer.

Return value

Type: [HRESULT](#)

Returns one of the [Direct3D 11 return codes](#).

Remarks

D3DGetBlobPart retrieves the part of a blob (arbitrary length data buffer) that contains the type of data that the *Part* parameter specifies.

Requirements

Target Platform	Windows
Header	d3dcompiler.h
Library	D3DCompiler.lib
DLL	D3DCompiler_47.dll

See also

[Functions](#)

D3DGetDebugInfo function (d3dcompiler.h)

Article • 10/13/2021 2 minutes to read

Note You can use this API to develop your Windows Store apps, but you can't use it in apps that you submit to the Windows Store.

Gets shader debug information.

Syntax

C++

```
HRESULT D3DGetDebugInfo(
    [in]    LPCVOID    pSrcData,
    [in]    SIZE_T     SrcDataSize,
    [out]   ID3DBlob **ppDebugInfo
);
```

Parameters

[in] `pSrcData`

Type: [LPCVOID](#)

A pointer to source data; either uncompiled or compiled HLSL code.

[in] `SrcDataSize`

Type: [SIZE_T](#)

Length of `pSrcData`.

[out] `ppDebugInfo`

Type: [ID3DBlob**](#)

A pointer to a buffer that receives the [ID3DBlob](#) interface that contains debug information.

Return value

Type: [HRESULT](#)

Returns one of the [Direct3D 11 return codes](#).

Remarks

Debug information is embedded in the body of the shader after calling [D3DCompile](#).

Requirements

Target Platform	Windows
Header	d3dcompiler.h
Library	D3dcompiler_47.lib
DLL	D3dcompiler_47.dll

See also

[Functions](#)

D3DGetInputAndOutputSignatureBlob function (d3dcompiler.h)

Article 10/13/2021

Note `D3DGetInputAndOutputSignatureBlob` may be altered or unavailable for releases after Windows 8.1. Instead use `D3DGetBlobPart` with the `D3D_BLOB_INPUT_AND_OUTPUT_SIGNATURE_BLOB` value.

Gets the input and output signatures from a compilation result.

Syntax

C++

```
HRESULT D3DGetInputAndOutputSignatureBlob(
    [in]    LPCVOID    pSrcData,
    [in]    SIZE_T     SrcDataSize,
    [out]   ID3DBlob **ppSignatureBlob
);
```

Parameters

[in] `pSrcData`

Type: [LPCVOID](#)

A pointer to source data as compiled HLSL code.

[in] `SrcDataSize`

Type: [SIZE_T](#)

Length of `pSrcData`.

[out] `ppSignatureBlob`

Type: [ID3DBlob**](#)

A pointer to a buffer that receives the [ID3DBlob](#) interface that contains a compiled shader.

Return value

Type: [HRESULT](#)

Returns one of the [Direct3D 11 return codes](#).

Requirements

Target Platform	Windows
Header	d3dcompiler.h
Library	D3dcompiler_47.lib
DLL	D3dcompiler_47.dll

See also

[Functions](#)

D3DGetInputSignatureBlob function (d3dcompiler.h)

Article • 10/13/2021 2 minutes to read

Note `D3DGetInputSignatureBlob` may be altered or unavailable for releases after Windows 8.1. Instead use `D3DGetBlobPart` with the `D3D_BLOB_INPUT_SIGNATURE_BLOB` value.

Gets the input signature from a compilation result.

Syntax

C++

```
HRESULT D3DGetInputSignatureBlob(
    [in]    LPCVOID    pSrcData,
    [in]    SIZE_T     SrcDataSize,
    [out]   ID3DBlob **ppSignatureBlob
);
```

Parameters

[in] `pSrcData`

Type: [LPCVOID](#)

A pointer to source data as compiled HLSL code.

[in] `SrcDataSize`

Type: [SIZE_T](#)

Length of `pSrcData`.

[out] `ppSignatureBlob`

Type: [ID3DBlob**](#)

A pointer to a buffer that receives the [ID3DBlob](#) interface that contains a compiled shader.

Return value

Type: [HRESULT](#)

Returns one of the [Direct3D 11 return codes](#).

Requirements

Target Platform	Windows
Header	d3dcompiler.h
Library	D3dcompiler_47.lib
DLL	D3dcompiler_47.dll

See also

[Functions](#)

D3DGetOutputSignatureBlob function (d3dcompiler.h)

Article • 10/13/2021 2 minutes to read

Note `D3DGetOutputSignatureBlob` may be altered or unavailable for releases after Windows 8.1. Instead use `D3DGetBlobPart` with the `D3D_BLOB_OUTPUT_SIGNATURE_BLOB` value.

Gets the output signature from a compilation result.

Syntax

C++

```
HRESULT D3DGetOutputSignatureBlob(
    [in]    LPCVOID    pSrcData,
    [in]    SIZE_T     SrcDataSize,
    [out]   ID3DBlob **ppSignatureBlob
);
```

Parameters

[in] `pSrcData`

Type: [LPCVOID](#)

A pointer to source data as compiled HLSL code.

[in] `SrcDataSize`

Type: [SIZE_T](#)

Length of `pSrcData`.

[out] `ppSignatureBlob`

Type: [ID3DBlob**](#)

A pointer to a buffer that receives the [ID3DBlob](#) interface that contains a compiled shader.

Return value

Type: [HRESULT](#)

Returns one of the [Direct3D 11 return codes](#).

Requirements

Target Platform	Windows
Header	d3dcompiler.h
Library	D3dcompiler_47.lib
DLL	D3dcompiler_47.dll

See also

[Functions](#)

D3DGetTraceInstructionOffsets function (d3dcompiler.h)

Article 10/13/2021

Retrieves the byte offsets for instructions within a section of shader code.

Syntax

C++

```
HRESULT D3DGetTraceInstructionOffsets(
    [in]          LPCVOID pSrcData,
    [in]          SIZE_T  SrcDataSize,
    [in]          UINT    Flags,
    [in]          SIZE_T  StartInstIndex,
    [in]          SIZE_T  NumInsts,
    [out, optional] SIZE_T *pOffsets,
    [out, optional] SIZE_T *pTotalInsts
);
```

Parameters

[in] *pSrcData*

A pointer to the compiled shader data.

[in] *SrcDataSize*

The size, in bytes, of the block of memory that *pSrcData* points to.

[in] *Flags*

A combination of the following flags that are combined by using a bitwise **OR** operation. The resulting value specifies how **D3DGetTraceInstructionOffsets** retrieves the instruction offsets.

Flag	Description
D3D_GET_INST_OFFSETS_INCLUDE_NON_EXECUTABLE (0x01)	Include non-executable code in the retrieved information.

[in] *StartInstIndex*

The index of the instruction in the compiled shader data for which **D3DGetTraceInstructionOffsets** starts to retrieve the byte offsets.

[in] NumInsts

The number of instructions for which **D3DGetTraceInstructionOffsets** retrieves the byte offsets.

[out, optional] pOffsets

A pointer to a variable that receives the actual number of offsets.

[out, optional] pTotalInsts

A pointer to a variable that receives the total number of instructions in the section of shader code.

Return value

Returns one of the [Direct3D 11 return codes](#).

Remarks

A new kind of Microsoft High Level Shader Language (HLSL) debugging information from a program database (PDB) file uses instruction-byte offsets within a shader blob (arbitrary-length data buffer). You use **D3DGetTraceInstructionOffsets** to translate to and from instruction indexes.

Note The D3dcompiler_44.dll or later version of the file contains the **D3DGetTraceInstructionOffsets** compiler function.

Requirements

Target Platform	Windows
Header	d3dcompiler.h
Library	D3DCompiler.lib

DLL

D3DCompiler_47.dll

See also

[Functions](#)

D3DLoadModule function (d3dcompiler.h)

Article • 10/13/2021 2 minutes to read

Creates a shader module interface from source data for the shader module.

Note This function is part of the HLSL shader linking technology that you can use on all Direct3D 11 platforms to create precompiled HLSL functions, package them into libraries, and link them into full shaders at run time.

Syntax

C++

```
HRESULT D3DLoadModule(  
    [in]    LPCVOID      pSrcData,  
    [in]    SIZE_T       cbSrcDataSize,  
    [out]   ID3D11Module **ppModule  
) ;
```

Parameters

[in] `pSrcData`

Type: [LPCVOID](#)

A pointer to the source data for the shader module.

[in] `cbSrcDataSize`

Type: [SIZE_T](#)

The size, in bytes, of the block of memory that `pSrcData` points to.

[out] `ppModule`

Type: [ID3D11Module**](#)

A pointer to a variable that receives a pointer to the [ID3D11Module](#) interface that is used for shader resource re-binding.

Return value

Type: [HRESULT](#)

Returns S_OK if successful; otherwise, returns one of the [Direct3D 11 Return Codes](#).

Remarks

Note The D3dcompiler_47.dll or later version of the DLL contains the **D3DLoadModule** function.

Requirements

Target Platform	Windows
Header	d3dcompiler.h
Library	D3DCompiler.lib
DLL	D3DCompiler_47.dll

See also

[Functions](#)

[ID3D11Module](#)

D3DPreprocess function (d3dcompiler.h)

Article • 10/13/2012 minutes to read

Preprocesses uncompiled HLSL code.

Syntax

C++

```
HRESULT D3DPreprocess(
    [in]          LPCVOID           pSrcData,
    [in]          SIZE_T            SrcDataSize,
    [in, optional] LPCSTR            pSourceName,
    [in, optional] const D3D_SHADER_MACRO *pDefines,
    [in, optional] ID3DInclude      *pInclude,
    [out]         ID3DBlob          **ppCodeText,
    [out, optional] ID3DBlob        **ppErrorMsgs
);
```

Parameters

[in] `pSrcData`

Type: [LPCVOID](#)

A pointer to uncompiled shader data; either ASCII HLSL code or a compiled effect.

[in] `SrcDataSize`

Type: [SIZE_T](#)

Length of `pSrcData`.

[in, optional] `pSourceName`

Type: [LPCSTR](#)

The name of the file that contains the uncompiled HLSL code.

[in, optional] `pDefines`

Type: [const D3D_SHADER_MACRO*](#)

An array of NULL-terminated macro definitions (see [D3D_SHADER_MACRO](#)).

[in, optional] pInclude

Type: [ID3DInclude*](#)

A pointer to an [ID3DInclude](#) for handling include files. Setting this to **NULL** will cause a compile error if a shader contains a #include. You can pass the **D3D_COMPILE_STANDARD_FILE_INCLUDE** macro, which is a pointer to a default include handler. This default include handler includes files that are relative to the current directory and files that are relative to the directory of the initial source file. When you use **D3D_COMPILE_STANDARD_FILE_INCLUDE**, you must specify the source file name in the *pSourceName* parameter; the compiler will derive the initial relative directory from *pSourceName*.

C++

```
#define D3D_COMPILE_STANDARD_FILE_INCLUDE ((ID3DInclude*)(UINT_PTR)1)
```

[out] ppCodeText

Type: [ID3DBlob**](#)

The address of a [ID3DBlob](#) that contains the compiled code.

[out, optional] ppErrorMsgs

Type: [ID3DBlob**](#)

A pointer to an [ID3DBlob](#) that contains compiler error messages, or **NULL** if there were no errors.

Return value

Type: [HRESULT](#)

Returns one of the [Direct3D 11 return codes](#).

Remarks

D3DPreprocess outputs **#line** directives and preserves line numbering of source input so that output line numbering can be properly related to the input source.

Requirements

Target Platform	Windows
Header	d3dcompiler.h
Library	D3dcompiler_47.lib
DLL	D3dcompiler_47.dll

See also

[Functions](#)

D3DReadFileToBlob function (d3dcompiler.h)

Article • 10/13/2021 2 minutes to read

Note You can use this API to develop your Windows Store apps, but you can't use it in apps that you submit to the Windows Store.

Reads a file that is on disk into memory.

Syntax

C++

```
HRESULT D3DReadFileToBlob(
    [in]  LPCWSTR  pFileName,
    [out] ID3DBlob **ppContents
);
```

Parameters

[in] `pFileName`

A pointer to a constant null-terminated string that contains the name of the file to read into memory.

[out] `ppContents`

A pointer to a variable that receives a pointer to the [ID3DBlob](#) interface that contains information that **D3DReadFileToBlob** read from the *pFileName* file. You can use this **ID3DBlob** interface to access the file information and pass it to other compiler functions.

Return value

Returns one of the [Direct3D 11 return codes](#).

Remarks

Note The D3dcompiler_44.dll or later version of the file contains the **D3DReadFileToBlob** compiler function.

Requirements

Target Platform	Windows
Header	d3dcompiler.h
Library	D3DCompiler.lib
DLL	D3DCompiler_47.dll

See also

[Functions](#)

D3DReflect function (d3dcompiler.h)

Article • 10/13/2021 2 minutes to read

Gets a pointer to a reflection interface.

Syntax

C++

```
HRESULT D3DReflect(  
    [in]    LPCVOID pSrcData,  
    [in]    SIZE_T   SrcDataSize,  
    [in]    REFIID   pInterface,  
    [out]   void     **ppReflector  
) ;
```

Parameters

[in] *pSrcData*

Type: [LPCVOID](#)

A pointer to source data as compiled HLSL code.

[in] *SrcDataSize*

Type: [SIZE_T](#)

Length of *pSrcData*.

[in] *pInterface*

Type: [REFIID](#)

The reference GUID of the COM interface to use. For example,
[IID_ID3D11ShaderReflection](#).

[out] *ppReflector*

Type: [void**](#)

A pointer to a reflection interface.

Return value

Type: [HRESULT](#)

Returns one of the [Direct3D 11 return codes](#).

Remarks

Shader code contains metadata that can be inspected using the reflection APIs.

The following code illustrates retrieving a [ID3D11ShaderReflection](#) Interface from a shader.

C++

```
pd3dDevice->CreatePixelShader( pPixelShaderBuffer->GetBufferPointer(),
                                 pPixelShaderBuffer->GetBufferSize(),
                                 g_pPSClassLinkage, &g_pPixelShader );

ID3D11ShaderReflection* pReflector = NULL;
D3DReflect( pPixelShaderBuffer->GetBufferPointer(), pPixelShaderBuffer-
            >GetBufferSize(),
            IID_ID3D11ShaderReflection, (void**) &pReflector);
```

Requirements

Target Platform	Windows
Header	d3dcompiler.h
Library	D3dcompiler_47.lib
DLL	D3dcompiler_47.dll

See also

[Functions](#)

D3DReflectLibrary function (d3dcompiler.h)

Article 10/13/2021

Creates a library-reflection interface from source data that contains an HLSL library of functions.

Note This function is part of the HLSL shader linking technology that you can use on all Direct3D 11 platforms to create precompiled HLSL functions, package them into libraries, and link them into full shaders at run time.

Syntax

C++

```
HRESULT D3DReflectLibrary(
    [in]  LPVOID pSrcData,
    [in]  SIZE_T SrcDataSize,
    [in]  REFIID riid,
    [out] LPVOID *ppReflector
);
```

Parameters

[in] *pSrcData*

Type: [LPCVOID](#)

A pointer to source data as an HLSL library of functions.

[in] *SrcDataSize*

Type: [SIZE_T](#)

The size, in bytes, of the block of memory that *pSrcData* points to.

[in] *riid*

Type: [REFIID](#)

The reference GUID of the COM interface to use. For example, [IID_ID3D11LibraryReflection](#).

[out] ppReflector

Type: [LPVOID*](#)

A pointer to a variable that receives a pointer to a library-reflection interface, [ID3D11LibraryReflection](#).

Return value

Type: [HRESULT](#)

Returns [S_OK](#) if successful; otherwise, returns one of the [Direct3D 11 Return Codes](#).

Requirements

Target Platform	Windows
Header	d3dcompiler.h
Library	D3DCompiler.lib
DLL	D3DCompiler_47.dll

See also

[Functions](#)

[ID3D11LibraryReflection](#)

D3DSetBlobPart function (d3dcompiler.h)

Article 10/13/2021

Sets information in a compilation result.

Syntax

C++

```
HRESULT D3DSetBlobPart(
    [in]    LPCVOID      pSrcData,
    [in]    SIZE_T       SrcDataSize,
    [in]    D3D_BLOB_PART Part,
    [in]    UINT          Flags,
    [in]    LPCVOID      pPart,
    [in]    SIZE_T       PartSize,
    [out]   ID3DBlob    **ppNewShader
);
```

Parameters

[in] *pSrcData*

Type: [LPCVOID](#)

A pointer to compiled shader data.

[in] *SrcDataSize*

Type: [SIZE_T](#)

The length of the compiled shader data that *pSrcData* points to.

[in] *Part*

Type: [D3D_BLOB_PART](#)

A [D3D_BLOB_PART](#)-typed value that specifies the part to set. Currently, you can update only private data; that is, **D3DSetBlobPart** currently only supports the [D3D_BLOB_PRIVATE_DATA](#) value.

[in] *Flags*

Type: [UINT](#)

Flags that indicate how to set the blob part. Currently, no flags are defined; therefore, set to zero.

[in] `pPart`

Type: [LPCVOID](#)

A pointer to data to set in the compilation result.

[in] `PartSize`

Type: [SIZE_T](#)

The length of the data that `pPart` points to.

[out] `ppNewShader`

Type: [ID3DBlob**](#)

A pointer to a buffer that receives the [ID3DBlob](#) interface for the new shader in which the new part data is set.

Return value

Type: [HRESULT](#)

Returns one of the [Direct3D 11 return codes](#).

Remarks

`D3DSetBlobPart` modifies data in a compiled shader. Currently, `D3DSetBlobPart` can update only the private data in a compiled shader. You can use `D3DSetBlobPart` to attach arbitrary uninterpreted data to a compiled shader.

Note The D3dcompiler_44.dll or later version of the file contains the `D3DSetBlobPart` compiler function.

Requirements

Target Platform	Windows
Header	d3dcompiler.h
Library	D3DCompiler.lib
DLL	D3DCompiler_47.dll

See also

[Functions](#)

D3DStripShader function (d3dcompiler.h)

Article • 10/13/2021 2 minutes to read

Removes unwanted blobs from a compilation result.

Syntax

C++

```
HRESULT D3DStripShader(
    [in]    LPCVOID    pShaderBytecode,
    [in]    SIZE_T     BytecodeLength,
    [in]    UINT       uStripFlags,
    [out]   ID3DBlob **ppStrippedBlob
);
```

Parameters

[in] `pShaderBytecode`

Type: [LPCVOID](#)

A pointer to source data as compiled HLSL code.

[in] `BytecodeLength`

Type: [SIZE_T](#)

Length of `pSrcData`.

[in] `uStripFlags`

Type: [UINT](#)

Strip flag options, represented by [D3DCOMPILER_STRIP_FLAGS](#).

[out] `ppStrippedBlob`

Type: [ID3DBlob**](#)

A pointer to a variable that receives a pointer to the [ID3DBlob](#) interface that you can use to access the unwanted stripped out shader code.

Return value

Type: [HRESULT](#)

Returns one of the [Direct3D 11 return codes](#).

Requirements

Target Platform	Windows
Header	d3dcompiler.h
Library	D3dcompiler_47.lib
DLL	D3dcompiler_47.dll

See also

[Functions](#)

D3DWriteBlobToFile function (d3dcompiler.h)

Article • 10/13/2021 2 minutes to read

Note You can use this API to develop your Windows Store apps, but you can't use it in apps that you submit to the Windows Store.

Writes a memory blob to a file on disk.

Syntax

C++

```
HRESULT D3DWriteBlobToFile(
    [in] ID3DBlob *pBlob,
    [in] LPCWSTR pFileName,
    [in] BOOL     bOverwrite
);
```

Parameters

[in] `pBlob`

Type: [ID3DBlob*](#)

A pointer to a [ID3DBlob](#) interface that contains the memory blob to write to the file that the *pFileName* parameter specifies.

[in] `pFileName`

Type: [LPCWSTR](#)

A pointer to a constant null-terminated string that contains the name of the file to which to write.

[in] `bOverwrite`

Type: [BOOL](#)

A Boolean value that specifies whether to overwrite information in the *pFileName* file. TRUE specifies to overwrite information and FALSE specifies not to overwrite information.

Return value

Type: [HRESULT](#)

Returns one of the [Direct3D 11 return codes](#).

Remarks

Note The D3dcompiler_44.dll or later version of the file contains the `D3DWriteBlobToFile` compiler function.

Requirements

Target Platform	Windows
Header	d3dcompiler.h
Library	D3DCompiler.lib
DLL	D3DCompiler_47.dll

See also

[Functions](#)

Structures (HLSL reference)

Article • 06/11/2020 • 2 minutes to read

This section contains information about the following Direct3D HLSL compiler structures:

Structure	Description
D3D_SHADER_DATA	Describes shader data.

Related topics

[D3DCompiler Reference](#)

D3D_SHADER_DATA structure (d3dcompiler.h)

Article • 07/27/2022 2 minutes to read

Describes shader data.

Syntax

C++

```
typedef struct _D3D_SHADER_DATA {
    LPCVOID pBytecode;
    SIZE_T BytecodeLength;
} D3D_SHADER_DATA;
```

Members

pBytecode

A pointer to shader data.

BytecodeLength

Length of shader data that **pBytecode** points to.

Remarks

An array of **D3D_SHADER_DATA** structures is passed to [D3DCompressShaders](#) to compress the shader data into a more compact form.

Requirements

Header

d3dcompiler.h

See also

Structures

Enumerations (HLSL reference)

Article • 08/19/2020 • 2 minutes to read

This section contains information about the following Direct3D HLSL compiler enumerations:

Enumeration	Description
D3D_BLOB_PART	Identifies parts of a blob (arbitrary length data buffer).
D3DCOMPILER_STRIP_FLAGS	Strip flag options.

Related topics

[D3DCompiler Reference](#)

D3D_BLOB_PART enumeration (d3dcompiler.h)

Article • 01/31/2022 2 minutes to read

Values that identify parts of the content of an arbitrary length data buffer.

Syntax

C++

```
typedef enum D3D_BLOB_PART {
    D3D_BLOB_INPUT_SIGNATURE_BLOB,
    D3D_BLOB_OUTPUT_SIGNATURE_BLOB,
    D3D_BLOB_INPUT_AND_OUTPUT_SIGNATURE_BLOB,
    D3D_BLOB_PATCH_CONSTANT_SIGNATURE_BLOB,
    D3D_BLOB_ALL_SIGNATURE_BLOB,
    D3D_BLOB_DEBUG_INFO,
    D3D_BLOB_LEGACY_SHADER,
    D3D_BLOB_XNA_PREPASS_SHADER,
    D3D_BLOB_XNA_SHADER,
    D3D_BLOB_PDB,
    D3D_BLOB_PRIVATE_DATA,
    D3D_BLOB_ROOT_SIGNATURE,
    D3D_BLOB_DEBUG_NAME,
    D3D_BLOB_TEST_ALTERNATE_SHADER = 0x8000,
    D3D_BLOB_TEST_COMPILE_DETAILS,
    D3D_BLOB_TEST_COMPILE_PERF,
    D3D_BLOB_TEST_COMPILE_REPORT
} ;
```

Constants

`D3D_BLOB_INPUT_SIGNATURE_BLOB`

The blob part is an input signature.

`D3D_BLOB_OUTPUT_SIGNATURE_BLOB`

The blob part is an output signature.

`D3D_BLOB_INPUT_AND_OUTPUT_SIGNATURE_BLOB`

The blob part is an input and output signature.

`D3D_BLOB_PATCH_CONSTANT_SIGNATURE_BLOB`

The blob part is a patch constant signature.

D3D_BLOB_ALL_SIGNATURE_BLOB

The blob part is all signature.

D3D_BLOB_DEBUG_INFO

The blob part is debug information.

D3D_BLOB_LEGACY_SHADER

The blob part is a legacy shader.

D3D_BLOB_XNA_PREPASS_SHADER

The blob part is an XNA preprocess shader.

D3D_BLOB_XNA_SHADER

The blob part is an XNA shader.

D3D_BLOB_PDB

The blob part is program database (PDB) information.

Note This value is supported by the D3dcompiler_44.dll or later version of the file.

D3D_BLOB_PRIVATE_DATA

The blob part is private data.

Note This value is supported by the D3dcompiler_44.dll or later version of the file.

D3D_BLOB_ROOT_SIGNATURE

The blob part is a root signature. Refer to [Specifying Root Signatures in HLSL](#) for more information on using Direct3D12 with HLSL.

Note This value is supported by the D3dcompiler_47.dll or later version of the file.

D3D_BLOB_DEBUG_NAME

The blob part is the debug name of the shader. If the application does not specify the debug name itself, an auto-generated name matching the PDB file of the shader is provided instead.

Note This value is supported by the D3dcompiler_47.dll as available on the Windows 10 Fall Creators Update and its SDK, or later version of the file.

D3D_BLOB_TEST_ALTERNATE_SHADER

Value: *0x8000*

The blob part is a test alternate shader.

Note This value identifies a test part and is only produced by special compiler versions. Therefore, this part type is typically not present in shaders.

D3D_BLOB_TEST_COMPILE_DETAILS

The blob part is test compilation details.

Note This value identifies a test part and is only produced by special compiler versions. Therefore, this part type is typically not present in shaders.

D3D_BLOB_TEST_COMPILE_PERF

The blob part is test compilation performance.

Note This value identifies a test part and is only produced by special compiler versions. Therefore, this part type is typically not present in shaders.

D3D_BLOB_TEST_COMPILE_REPORT

The blob part is a test compilation report.

Note This value identifies a test part and is only produced by special compiler versions. Therefore, this part type is typically not present in shaders.

Note This value is supported by the D3dcompiler_44.dll or later version of the file.

Remarks

These values are passed to the [D3DGetBlobPart](#) or [D3DSetBlobPart](#) function.

Requirements

Header	d3dcompiler.h
--------	---------------

See also

[Enumerations](#)

D3DCOMPILER_STRIP_FLAGS enumeration (d3dcompiler.h)

Article • 01/31/2022 2 minutes to read

Strip flag options.

Syntax

C++

```
typedef enum D3DCOMPILER_STRIP_FLAGS {
    D3DCOMPILER_STRIP_REFLECTION_DATA = 0x00000001,
    D3DCOMPILER_STRIP_DEBUG_INFO = 0x00000002,
    D3DCOMPILER_STRIP_TEST_BLOBS = 0x00000004,
    D3DCOMPILER_STRIP_PRIVATE_DATA = 0x00000008,
    D3DCOMPILER_STRIP_ROOT_SIGNATURE = 0x00000010,
    D3DCOMPILER_STRIP_FORCE_DWORD = 0x7fffffff
} ;
```

Constants

D3DCOMPILER_STRIP_REFLECTION_DATA

Value: *0x00000001*

Remove reflection data.

D3DCOMPILER_STRIP_DEBUG_INFO

Value: *0x00000002*

Remove debug information.

D3DCOMPILER_STRIP_TEST_BLOBS

Value: *0x00000004*

Remove test blob data.

D3DCOMPILER_STRIP_PRIVATE_DATA

Value: 0x00000008

Note This value is supported by the D3dcompiler_44.dll or later version of the file.

Remove private data.

D3DCOMPILER_STRIP_ROOT_SIGNATURE

Value: 0x00000010

Note This value is supported by the D3dcompiler_47.dll or later version of the file.

Remove the root signature. Refer to [Specifying Root Signatures in HLSL](#) for more information on using Direct3D12 with HLSL.

D3DCOMPILER_STRIP_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

These flags are used by [D3DStripShader](#).

Requirements

Header

d3dcompiler.h

See also

[Enumerations](#)

D3DCompiler Constants

Article • 11/20/2019 • 2 minutes to read

This section contains information about the following Direct3D HLSL compiler constants:

Enumeration	Description
D3DCOMPILE Constants	These constants are shader compiler options.
D3DCOMPILE_EFFECT Constants	These constants are effect compiler options.

Related topics

[D3DCompiler Reference](#)

D3DCOMPILE Constants

Article • 02/13/2022 • 3 minutes to read

The D3DCOMPILE constants specify how the compiler compiles the HLSL code.

Constant	Description	Note
D3DCOMPILE_DEBUG /Zi	Directs the compiler to insert debug file/line/type/symbol information into the output code.	See D3DXSHADER_DEBUG
D3DCOMPILE_SKIP_VALIDATION /Vd	Directs the compiler not to validate the generated code against known capabilities and constraints. We recommend that you use this constant only with shaders that have been successfully compiled in the past. DirectX always validates shaders before it sets them to a device.	See D3DXSHADER_SKIPVALIDATION
D3DCOMPILE_SKIP_OPTIMIZATION /Od	Directs the compiler to skip optimization steps during code generation. We recommend that you set this constant for debug purposes only.	See D3DXSHADER_SKIPOPTIMIZATION
D3DCOMPILE_PACK_MATRIX_ROW_MAJOR /Zpr	Directs the compiler to pack matrices in row-major order on input and output from the shader.	See D3DXSHADER_PACKMATRIX_ROWMAJOR
D3DCOMPILE_PACK_MATRIX_COLUMN_MAJOR /Zpc	Directs the compiler to pack matrices in column-major order on input and output from the shader. This type of packing is generally more efficient because a series of dot-products can then perform vector-matrix multiplication.	See D3DXSHADER_PACKMATRIX_COLUMNMAJOR

Constant	Description	Note
D3DCOMPILE_PARTIAL_PRECISION /Gpp	Directs the compiler to perform all computations with partial precision. If you set this constant, the compiled code might run faster on some hardware.	See D3DXSHADER_PARTIALPRECISION
D3DCOMPILE_FORCE_VS_SOFTWARE_NO_OPT	Directs the compiler to compile a vertex shader for the next highest shader profile. This constant turns debugging on and optimizations off.	This flag was applicable only to Direct3D 9. See D3DXSHADER_FORCE_VS_SOFTWARE_NOOPT
D3DCOMPILE_FORCE_PS_SOFTWARE_NO_OPT	Directs the compiler to compile a pixel shader for the next highest shader profile. This constant also turns debugging on and optimizations off.	This flag was applicable only to Direct3D 9. See D3DXSHADER_FORCE_PS_SOFTWARE_NOOPT
D3DCOMPILE_NO_PRESHADER /Op	Directs the compiler to disable Preshaders. If you set this constant, the compiler does not pull out static expression for evaluation.	This flag was only applicable to legacy Direct3D 9 and Direct3D 10 Effects (FX). See D3DXSHADER_NO_PRESHADER
D3DCOMPILE_AVOID_FLOW_CONTROL /Gfa	Directs the compiler to not use flow-control constructs where possible.	See D3DXSHADER_AVOID_FLOW_CONTROL
D3DCOMPILE_ENABLE_STRICTNESS /Ges	Forces strict compile, which might not allow for legacy syntax. By default, the compiler disables strictness on deprecated syntax.	
D3DCOMPILE_IEEE_STRICTNESS /Gis	Forces the IEEE strict compile which avoids optimizations that may break IEEE rules.	See D3DXSHADER_IEEE_STRICTNESS
D3DCOMPILE_ENABLE_BACKWARDS_COMPATIBILITY /Gec	Directs the compiler to enable older shaders to compile to 5_0 targets.	See D3DXSHADER_ENABLE_BACKWARDS_COMPATIBILITY

Constant	Description	Note
D3DCOMPILE_OPTIMIZATION_LEVEL0 /O0	<p>Directs the compiler to use the lowest optimization level. If you set this constant, the compiler might produce slower code but produces the code more quickly. Set this constant when you develop the shader iteratively.</p>	See D3DXSHADER_OPTIMIZATION_LEVEL0
D3DCOMPILE_OPTIMIZATION_LEVEL1 /O1	<p>Directs the compiler to use the second lowest optimization level.</p>	See D3DXSHADER_OPTIMIZATION_LEVEL1
D3DCOMPILE_OPTIMIZATION_LEVEL2 /O2	<p>Directs the compiler to use the second highest optimization level.</p>	See D3DXSHADER_OPTIMIZATION_LEVEL2
D3DCOMPILE_OPTIMIZATION_LEVEL3 /O3	<p>Directs the compiler to use the highest optimization level. If you set this constant, the compiler produces the best possible code but might take significantly longer to do so. Set this constant for final builds of an application when performance is the most important factor.</p>	See D3DXSHADER_OPTIMIZATION_LEVEL3
D3DCOMPILE_WARNINGS_ARE_ERRORS /WX	<p>Directs the compiler to treat all warnings as errors when it compiles the shader code. We recommend that you use this constant for new shader code, so that you can resolve all warnings and lower the number of hard-to-find code defects.</p>	
D3DCOMPILE_RESOURCES_MAY_ALIAS /res_may_alias	<p>Directs the compiler to assume that unordered access views (UAVs) and shader resource views (SRVs) may alias for cs_5_0.</p>	Only applies to DirectX 12 / Shader Model 5.1

Constant	Description	Note
D3DCOMPILER_ENABLE_UNBOUNDED_DESCRIPTOR_TABLES /enable_unbounded_descriptor_tables	Directs the compiler to enable unbounded descriptor tables.	Only applies to DirectX 12 / Shader Model 5.1
D3DCOMPILER_ALL_RESOURCES_BOUND /all_resources_bound	Directs the compiler to ensure all resources are bound.	Only applies to DirectX 12 / Shader Model 5.1
D3DCOMPILER_DEBUG_NAME_FOR_SOURCE /Zss	When generating debug PDBs this makes use of the source file and binary for the hash.	
D3DCOMPILER_DEBUG_NAME_FOR_BINARY /Zsb	When generating debug PDBs this makes use of the binary file name only for the hash.	

ⓘ Note

The D3DCOMPILER_RESOURCES_MAY_ALIAS, D3DCOMPILER_ENABLE_UNBOUNDED_DESCRIPTOR_TABLES, and D3DCOMPILER_ALL_RESOURCES_BOUND compiler constants are new starting with the D3dcompiler_47.dll that ships with the Windows 8.1 SDK or later.

ⓘ Note

The D3DCOMPILER_DEBUG_NAME_FOR_SOURCE and D3DCOMPILER_DEBUG_NAME_FOR_BINARY compiler constants are new starting with the D3dcompiler_47.dll that ships with the Windows 10 Fall Creator's Update SDK (version 16299) or later. See [this blog post](#).

ⓘ Note

For DirectX 12, Shader Model 5.1, the D3DCompile API, and FXC are all deprecated. Use Shader Model 6 via DXIL instead. See [GitHub](#).

Requirements

Requirement	Value
Header	D3DCompiler.h

See also

[D3DCompiler Constants](#)

D3DCOMPILE_EFFECT Constants

Article • 03/15/2021 • 2 minutes to read

These constants direct how the compiler compiles an effect file or how the runtime processes the effect file.

D3DCOMPILE_EFFECT_CHILD_EFFECT

($1 << 0$)

Compile the effects (.fx) file to a child effect. Child effects have no initializers for any shared values because these child effects are initialized in the master effect (the effect pool).

ⓘ Note

Effect pools are supported by Effects 10 (FX10) but not by Effects 11 (FX11). For more info about differences between effect pools in Direct3D 10 and effect groups in Direct3D 11, see [Effect Pools and Groups](#).

D3DCOMPILE_EFFECT_ALLOW_SLOW_OPS

($1 << 1$)

Disables performance mode and allows for mutable state objects.

By default, performance mode is enabled. Performance mode disallows mutable state objects by preventing non-literal expressions from appearing in state object definitions.

Requirements

Requirement	Value
Header	D3DCompiler.h

See also

[D3DCompiler Constants](#)

Inline Format Conversion Reference

Article • 04/09/2021 • 2 minutes to read

This section contains the following sections

- [Functions](#)
- [Structures](#)

The D3DX_DXGIFormatConvert.inl header ships in the legacy DirectX SDK and relied on XNAMath for C++ support. It is also included in the [Microsoft.DXSDK.D3DX](#) NuGet package. The latest version uses DirectXMath for C++ support, and all functions are defined in the [DirectX C++](#) namespace.

Related topics

[Reference for HLSL](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

Format conversion functions (HLSL reference)

Article • 04/09/2021 • 2 minutes to read

The section contains the format conversion functions used in Compute and Pixel Shaders.

- [Converter Functions](#)
- [Related topics](#)

The D3DX_DXGIFormatConvert.inl header ships in the legacy DirectX SDK and relied on XNAMath for C++ support. It is also included in the [Microsoft.DXSDK.D3DX](#) NuGet package. The latest version uses DirectXMath for C++ support, and all functions are defined in the [DirectX C++](#) namespace.

Converter Functions

DXGI_FORMAT_R10G10B10A2_UNORM

[D3DX_R10G10B10A2_UNORM_to_FLOAT4](#)
[D3DX_FLOAT4_to_R10G10B10A2_UNORM](#)

DXGI_FORMAT_R10G10B10A2_UINT

[D3DX_R10G10B10A2_UINT_to_UINT4](#)
[D3DX_UINT4_to_R10G10B10A2_UINT](#)

DXGI_FORMAT_R8G8B8A8_UNORM:

[D3DX_R8G8B8A8_UNORM_to_FLOAT4](#)
[D3DX_FLOAT4_to_R8G8B8A8_UNORM](#)

DXGI_FORMAT_R8G8B8A8_UNORM_SRGB

[D3DX_R8G8B8A8_UNORM_SRGB_to_FLOAT4_inexact](#)
[D3DX_R8G8B8A8_UNORM_SRGB_to_FLOAT4](#)
[D3DX_FLOAT4_to_R8G8B8A8_UNORM_SRGB](#)

DXGI_FORMAT_R8G8B8A8_UINT

[D3DX_R8G8B8A8_UINT_to_UINT4](#)
[D3DX_UINT4_to_R8G8B8A8_UINT](#)

DXGI_FORMAT_R8G8B8A8_SNORM

[D3DX_R8G8B8A8_SNORM_to_FLOAT4](#)
[D3DX_FLOAT4_to_R8G8B8A8_SNORM](#)

DXGI_FORMAT_R8G8B8A8_SINT

[D3DX_R8G8B8A8_SINT_to_INT4](#)
[D3DX_INT4_to_R8G8B8A8_SINT](#)

DXGI_FORMAT_B8G8R8A8_UNORM

[D3DX_B8G8R8A8_UNORM_to_FLOAT4](#)
[D3DX_FLOAT4_to_B8G8R8A8_UNORM](#)

DXGI_FORMAT_B8G8R8A8_UNORM_SRGB

[D3DX_B8G8R8A8_UNORM_SRGB_to_FLOAT4_inexact](#)
[D3DX_B8G8R8A8_UNORM_SRGB_to_FLOAT4](#)
[D3DX_FLOAT4_to_R8G8B8A8_UNORM_SRGB](#)

DXGI_FORMAT_B8G8R8X8_UNORM

[D3DX_B8G8R8X8_UNORM_to_FLOAT3](#)
[D3DX_FLOAT3_to_B8G8R8X8_UNORM](#)

DXGI_FORMAT_B8G8R8X8_UNORM_SRGB

[D3DX_B8G8R8X8_UNORM_SRGB_to_FLOAT3_inexact](#)
[D3DX_B8G8R8X8_UNORM_SRGB_to_FLOAT3](#)
[D3DX_FLOAT3_to_B8G8R8X8_UNORM_SRGB](#)

DXGI_FORMAT_R16G16_FLOAT

[D3DX_R16G16_FLOAT_to_FLOAT2](#)

[D3DX_FLOAT2_to_R16G16_FLOAT](#)

DXGI_FORMAT_R16G16_UNORM

[D3DX_R16G16_UNORM_to_FLOAT2](#)

[D3DX_FLOAT2_to_R16G16_UNORM](#)

DXGI_FORMAT_R16G16_UINT

[D3DX_R16G16_UINT_to_UINT2](#)

[D3DX_UINT2_to_R16G16_UINT](#)

DXGI_FORMAT_R16G16_SNORM

[D3DX_R16G16_SNORM_to_FLOAT2](#)

[D3DX_FLOAT2_to_R16G16_SNORM](#)

DXGI_FORMAT_R16G16_SINT

[D3DX_R16G16_SINT_to_INT2](#)

[D3DX_INT2_to_R16G16_SINT](#)

Related topics

[Inline Format Conversion Reference](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_B8G8R8A8_UNORM_SRGB_to_FLOAT4 function

Article • 03/15/2021 • 2 minutes to read

Unpacks DXGI_FORMAT_B8G8R8A8_UNORM_SRGB shader data to an XMFLOAT4.

Syntax

syntax

```
XMFLOAT4 D3DX_B8G8R8A8_UNORM_SRGB_to_FLOAT4(  
    UINT packedInput  
) ;
```

Parameters

packedInput

The packed shader data.

Return value

The unpacked shader data.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

[Functions](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_B8G8R8A8_UNORM_SRGB_to_FLOAT4_inexact function

Article • 03/15/2021 • 2 minutes to read

Unpacks DXGI_FORMAT_B8G8R8A8_UNORM_SRGB shader data to an XMFLOAT4.

Syntax

syntax

```
XMFLOAT4 D3DX_B8G8R8A8_UNORM_SRGB_to_FLOAT4_inexact(  
    UINT packedInput  
) ;
```

Parameters

packedInput

The packed shader data.

Return value

The unpacked shader data.

Remarks

This function uses shader instructions that don't have high enough precision to give the exact answer. The alternative function [D3DX_B8G8R8A8_UNORM_SRGB_to_FLOAT4](#) uses a lookup table stored in the shader to give an exact SRGB to float conversion.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

Functions

Unpacking and Packing DXGI_FORMAT for In-Place Image Editing

D3DX_B8G8R8A8_UNORM_to_FLOAT4 function

Article • 03/15/2021 • 2 minutes to read

Unpacks DXGI_FORMAT_B8G8R8A8_UNORM shader data to an XMFLOAT4.

Syntax

syntax

```
XMFLOAT4 D3DX_B8G8R8A8_UNORM_to_FLOAT4(  
    UINT packedInput  
) ;
```

Parameters

packedInput

The packed shader data.

Return value

The unpacked shader data.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

[Functions](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_B8G8R8X8_UNORM_SRGB_to_FLOAT3 function

Article • 03/15/2021 • 2 minutes to read

Unpacks DXGI_FORMAT_B8G8R8X8_UNORM_SRGB shader data to an XMFLOAT3.

Syntax

syntax

```
XMFLOAT3 D3DX_B8G8R8X8_UNORM_SRGB_to_FLOAT3(  
    UINT packedInput  
) ;
```

Parameters

packedInput

The packed shader data.

Return value

The unpacked shader data.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

[Functions](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_B8G8R8X8_UNORM_SRGB_to_FLOAT3_inexact function

Article • 03/15/2021 • 2 minutes to read

Unpacks DXGI_FORMAT_B8G8R8X8_UNORM_SRGB shader data to an XMFLOAT3.

Syntax

syntax

```
XMFLOAT3 D3DX_B8G8R8X8_UNORM_SRGB_to_FLOAT3_inexact(  
    UINT packedInput  
) ;
```

Parameters

packedInput

The packed shader data.

Return value

The unpacked shader data.

Remarks

This function uses shader instructions that don't have high enough precision to give the exact answer. The alternative function [D3DX_B8G8R8X8_UNORM_SRGB_to_FLOAT3](#) uses a lookup table stored in the shader to give an exact SRGB to float conversion.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

Functions

Unpacking and Packing DXGI_FORMAT for In-Place Image Editing

D3DX_B8G8R8X8_UNORM_to_FLOAT3 function

Article • 03/15/2021 • 2 minutes to read

Unpacks DXGI_FORMAT_B8G8R8X8_UNORM shader data to an XMFLOAT3.

Syntax

syntax

```
XMFLOAT3 D3DX_B8G8R8X8_UNORM_to_FLOAT3(  
    UINT packedInput  
) ;
```

Parameters

packedInput

The packed shader data.

Return value

The unpacked shader data.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

[Functions](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_FLOAT_to_INT function

Article • 03/15/2021 • 2 minutes to read

Converts a FLOAT value to INT.

Syntax

syntax

```
INT D3DX_FLOAT_to_INT(  
    FLOAT _V,  
    FLOAT _Scale  
) ;
```

Parameters

_V

The v value.

_Scale

The scale value.

Return value

The converted FLOAT value

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

[Functions](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_FLOAT_to_SRGB function

Article • 03/15/2021 • 2 minutes to read

Converts a FLOAT value to an SRGB.

Syntax

syntax

```
FLOAT D3DX_FLOAT_to_SRGB(  
    FLOAT val  
)
```

Parameters

val

FLOAT value to convert.

Return value

The converted SRGB value.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

[Functions](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_FLOAT_to_UINT function

Article • 03/15/2021 • 2 minutes to read

Converts a FLOAT value to UINT.

Syntax

syntax

```
UINT D3DX_FLOAT_to_UINT(  
    FLOAT _V,  
    FLOAT _Scale  
) ;
```

Parameters

_V

The v vector.

_Scale

The scale value.

Return value

The converted FLOAT value

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

[Functions](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_FLOAT2_to_R16G16_FLOAT function

Article • 03/15/2021 • 2 minutes to read

Packs the given XMFLOAT2 back into a DXGI_FORMAT_R16G16_FLOAT.

Syntax

syntax

```
UINT D3DX_FLOAT2_to_R16G16_FLOAT(
    XMFLOAT2 unpackedInput
);
```

Parameters

unpackedInput

The unpacked shader data.

Return value

The packed shader data.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

[Functions](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_FLOAT2_to_R16G16_SNORM function

Article • 03/15/2021 • 2 minutes to read

Packs the given XMFLOAT2 back into a DXGI_FORMAT_R16G16_SNORM.

Syntax

syntax

```
UINT D3DX_FLOAT2_to_R16G16_SNORM(
    hsls_precise XMFLOAT2 unpackedInput
);
```

Parameters

unpackedInput

The unpacked shader data.

Return value

The packed shader data.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

[Functions](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_FLOAT2_to_R16G16_UNORM function

Article • 03/15/2021 • 2 minutes to read

Packs the given XMFLOAT2 back into a DXGI_FORMAT_R16G16_UNORM.

Syntax

syntax

```
UINT D3DX_FLOAT2_to_R16G16_UNORM(
    XMFLOAT2 unpackedInput
);
```

Parameters

unpackedInput

The unpacked shader data.

Return value

The packed shader data.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

[Functions](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_FLOAT3_to_B8G8R8X8_UNORM function

Article • 03/15/2021 • 2 minutes to read

Packs the given XMFLOAT3 into a DXGI_FORMAT_B8G8R8X8_UNORM UINT.

Syntax

syntax

```
UINT D3DX_FLOAT3_to_B8G8R8X8_UNORM(  
    hsls1_precise XMFLOAT3 unpackedInput  
) ;
```

Parameters

unpackedInput

The unpacked shader data.

Return value

The packed shader data.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

Functions

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_FLOAT3_to_B8G8R8X8_UNORM_SRGB function

Article • 03/15/2021 • 2 minutes to read

Packs the given XMFLOAT3 back into a DXGI_FORMAT_B8G8R8X8_UNORM_SRGB.

Syntax

```
syntax  
  
UINT D3DX_FLOAT3_to_B8G8R8X8_UNORM_SRGB(  
    hsls1_precise XMFLOAT3 unpackedInput  
) ;
```

Parameters

unpackedInput

The shader data to pack.

Return value

The packed shader data.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

[Functions](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_FLOAT4_to_R10G10B10A2_UNORM function

Article • 03/15/2021 • 2 minutes to read

Packs the given XMFLOAT4 back into a DXGI_FORMAT_R10G10B10A2_UNORM.

Syntax

syntax

```
UINT D3DX_FLOAT4_to_R10G10B10A2_UNORM(
    hsls1_precise XMFLOAT4 unpackedInput
);
```

Parameters

unpackedInput

The shader data to pack.

Return value

The packed shader data.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

[Functions](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_FLOAT4_to_B8G8R8A8_UNORM function

Article • 03/15/2021 • 2 minutes to read

Packs the given XMFLOAT4 back into a DXGI_FORMAT_B8G8R8A8_UNORM.

Syntax

syntax

```
UINT D3DX_FLOAT4_to_B8G8R8A8_UNORM(  
    hsls1_precise XMFLOAT4 unpackedInput  
) ;
```

Parameters

unpackedInput

The shader data to pack.

Return value

The packed shader data.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

Functions

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_FLOAT4_to_B8G8R8A8_UNORM_SRGB function

Article • 03/15/2021 • 2 minutes to read

Packs the given XMFLOAT4 into a DXGI_FORMAT_B8G8R8A8_UNORM_SRGB UINT.

Syntax

```
syntax  
  
UINT D3DX_FLOAT4_to_B8G8R8A8_UNORM_SRGB(  
    hsls1_precise XMFLOAT4 unpackedInput  
) ;
```

Parameters

unpackedInput

The unpacked shader data.

Return value

The packed shader data.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

[Functions](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_FLOAT4_to_R8G8B8A8_SNORM function

Article • 03/15/2021 • 2 minutes to read

Packs the given XMFLOAT4 back into a DXGI_FORMAT_R8G8B8A8_SNORM.

Syntax

syntax

```
UINT D3DX_FLOAT4_to_R8G8B8A8_SNORM(  
    hsls1_precise XMFLOAT4 unpackedInput  
) ;
```

Parameters

unpackedInput

The shader data to pack.

Return value

The packed shader data.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

[Functions](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_FLOAT4_to_R8G8B8A8_UNORM function

Article • 03/15/2021 • 2 minutes to read

Unpacks DXGI_FORMAT_R8G8B8A8_UNORM shader data to an XMFLOAT4.

Syntax

syntax

```
XMFLOAT4 D3DX_FLOAT4_to_R8G8B8A8_UNORM(
    UINT packedInput
);
```

Parameters

packedInput

The packed shader data.

Return value

The unpacked shader data.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

[Functions](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_FLOAT4_to_R8G8B8A8_UNORM_SRGB function

Article • 03/15/2021 • 2 minutes to read

Packs the given XMFLOAT4 back into a DXGI_FORMAT_R8G8B8A8_UNORM_SRGB.

Syntax

```
syntax  
  
UINT D3DX_FLOAT4_to_R8G8B8A8_UNORM_SRGB(  
    hsls1_precise XMFLOAT4 unpackedInput  
);
```

Parameters

unpackedInput

The shader data to pack.

Return value

The packed shader data.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

[Functions](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_INT_to_FLOAT function

Article • 03/15/2021 • 2 minutes to read

Converts a INT value to FLOAT.

Syntax

syntax

```
FLOAT D3DX_INT_to_FLOAT(  
    INT _V,  
    FLOAT _Scale  
)
```

Parameters

_V

The v value.

_Scale

The scale value.

Return value

The converted int value.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

[Functions](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_INT2_to_R16G16_SINT function

Article • 03/15/2021 • 2 minutes to read

Packs the given XMINT2 back into a DXGI_FORMAT_R16G16_SINT.

Syntax

syntax

```
UINT D3DX_INT2_to_R16G16_SINT(  
    hsls_precise XMINT2 unpackedInput  
) ;
```

Parameters

unpackedInput

The shader data to pack.

Return value

The packed shader data.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

Functions

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_INT4_to_R8G8B8A8_SINT function

Article • 03/15/2021 • 2 minutes to read

Packs the given XMINT4 back into a DXGI_FORMAT_R8G8B8A8_SINT.

Syntax

syntax

```
UINT D3DX_INT4_to_R8G8B8A8_SINT(  
    hsls_precision XMINT4 unpackedInput  
) ;
```

Parameters

unpackedInput

The shader data to pack.

Return value

The packed shader data.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

Functions

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_IsNaN function

Article • 03/15/2021 • 2 minutes to read

Determines if the value is a NaN (Not a Number).

Syntax

syntax

```
bool D3DX_IsNaN(  
    FLOAT _V  
) ;
```

Parameters

_V

The specified value.

Return value

true if a NaN; otherwise false.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

Functions

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_R10G10B10A2_UINT_to_UINT4 function

Article • 03/15/2021 • 2 minutes to read

Unpacks DXGI_FORMAT_R10G10B10A2_UINT shader data to an XMINT4.

Syntax

syntax

```
XMFLOAT4 D3DX_R10G10B10A2_UINT_to_UINT4(  
    uint packedInput  
) ;
```

Parameters

packedInput

The packed shader data.

Return value

The unpacked shader data.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

[Functions](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_R10G10B10A2_UNORM_to_FLOAT4 function

Article • 03/15/2021 • 2 minutes to read

Unpacks DXGI_FORMAT_R10G10B10A2_UNORM shader data to an XMFLOAT4.

Syntax

```
syntax  
  
XMFLOAT4 D3DX_R10G10B10A2_UNORM_to_FLOAT4(  
    UINT packedInput  
) ;
```

Parameters

packedInput

The packed shader data.

Return value

The unpacked shader data.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

[Functions](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_R16G16_FLOAT_to_FLOAT2 function

Article • 03/15/2021 • 2 minutes to read

Unpacks DXGI_FORMAT_B8G8R8X8_UNORM_SRGB shader data to an XMFLOAT2.

Syntax

syntax

```
XMFLOAT2 D3DX_R16G16_FLOAT_to_FLOAT2(  
    UINT packedInput  
) ;
```

Parameters

packedInput

The packed shader data.

Return value

The unpacked shader data.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

[Functions](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_R16G16_SINT_to_INT2 function

Article • 03/15/2021 • 2 minutes to read

Unpacks DXGI_FORMAT_R16G16_SINT shader data to an XMINT2.

Syntax

syntax

```
XMINT2 D3DX_R16G16_SINT_to_INT2(  
    UINT packedInput  
) ;
```

Parameters

packedInput

The packed shader data.

Return value

The unpacked shader data.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

[Functions](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_R16G16_SNORM_to_FLOAT2 function

Article • 03/15/2021 • 2 minutes to read

Unpacks DXGI_FORMAT_R16G16_SNORM shader data to an XMFLOAT2.

Syntax

syntax

```
XMFLOAT2 D3DX_R16G16_SNORM_to_FLOAT2(  
    UINT packedInput  
) ;
```

Parameters

packedInput

The packed shader data.

Return value

The unpacked shader data.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

[Functions](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_R16G16_UINT_to_UINT2 function

Article • 03/15/2021 • 2 minutes to read

Unpacks DXGI_FORMAT_R16G16_UINT shader data to an XMUINT2.

Syntax

syntax

```
XMUINT2 D3DX_R16G16_UINT_to_UINT2(  
    UINT packedInput  
) ;
```

Parameters

packedInput

The packed shader data.

Return value

The unpacked shader data.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

Functions

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_R16G16_UNORM_to_FLOAT2 function

Article • 03/15/2021 • 2 minutes to read

Unpacks DXGI_FORMAT_R16G16_UNORM shader data to an XMFLOAT2.

Syntax

syntax

```
XMFLOAT2 D3DX_R16G16_UNORM_to_FLOAT2(  
    UINT packedInput  
) ;
```

Parameters

packedInput

The packed shader data.

Return value

The unpacked shader data.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

[Functions](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_R8G8B8A8_SINT_to_INT4 function

Article • 03/15/2021 • 2 minutes to read

Unpacks DXGI_FORMAT_R8G8B8A8_SINT shader data to an XMINT4.

Syntax

syntax

```
XMFLOAT4 D3DX_R8G8B8A8_SINT_to_INT4(  
    UINT packedInput  
) ;
```

Parameters

packedInput

The packed shader data.

Return value

The unpacked shader data.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

[Functions](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_R8G8B8A8_SNORM_to_FLOAT4 function

Article • 03/15/2021 • 2 minutes to read

Unpacks DXGI_FORMAT_R8G8B8A8_SNORM shader data to an XMFLOAT4.

Syntax

syntax

```
XMFLOAT4 D3DX_R8G8B8A8_SNORM_to_FLOAT4(  
    UINT packedInput  
) ;
```

Parameters

packedInput

The packed shader data.

Return value

The unpacked shader data.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

[Functions](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_R8G8B8A8_UINT_to_UINT4 function

Article • 03/15/2021 • 2 minutes to read

Unpacks DXGI_FORMAT_R8G8B8A8_UINT shader data to an XMUINT4.

Syntax

syntax

```
XMFLOAT4 D3DX_R8G8B8A8_UINT_to_UINT4(  
    uint packedInput  
) ;
```

Parameters

packedInput

The packed shader data.

Return value

The unpacked shader data.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

[Functions](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_R8G8B8A8_UNORM_SRGB_to_FLOAT4 function

Article • 03/15/2021 • 2 minutes to read

Unpacks DXGI_FORMAT_R8G8B8A8_UNORM_SRGB shader data to an XMFLOAT4.

Syntax

syntax

```
XMFLOAT4 D3DX_R8G8B8A8_UNORM_SRGB_to_FLOAT4(  
    UINT packedInput  
) ;
```

Parameters

packedInput

The packed shader data.

Return value

The unpacked shader data.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

[Functions](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_R8G8B8A8_UNORM_SRGB_to_FLOAT4_inexact function

Article • 03/15/2021 • 2 minutes to read

Unpacks DXGI_FORMAT_R8G8B8A8_UNORM_SRGB shader data to an XMFLOAT4.

Syntax

syntax

```
XMFLOAT4 D3DX_R8G8B8A8_UNORM_SRGB_to_FLOAT4_inexact(  
    UINT packedInput  
) ;
```

Parameters

packedInput

The packed shader data.

Return value

The unpacked shader data.

Remarks

This function uses shader instructions that don't have high enough precision to give the exact answer. The alternative function [D3DX_R8G8B8A8_UNORM_SRGB_to_FLOAT4](#) uses a lookup table stored in the shader to give an exact SRGB to float conversion.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

Functions

Unpacking and Packing DXGI_FORMAT for In-Place Image Editing

D3DX_R8G8B8A8_UNORM_to_FLOAT4 function

Article • 03/15/2021 • 2 minutes to read

Unpacks DXGI_FORMAT_R8G8B8A8_UNORM shader data to an XMFLOAT4.

Syntax

syntax

```
XMFLOAT4 D3DX_R8G8B8A8_UNORM_to_FLOAT4(  
    UINT packedInput  
) ;
```

Parameters

packedInput

The packed shader data.

Return value

The unpacked shader data.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

[Functions](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_Saturate_FLOAT function

Article • 03/15/2021 • 2 minutes to read

Retrieves the saturated value of the given FLOAT.

Syntax

syntax

```
FLOAT D3DX_Saturate_FLOAT(  
    FLOAT _V  
) ;
```

Parameters

_V

The value to saturate.

Return value

The saturated value.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

[Functions](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_SaturateSigned_FLOAT function

Article • 03/15/2021 • 2 minutes to read

Retrieves a signed saturated value from the given FLOAT.

Syntax

syntax

```
D3DX_SaturateSigned_FLOAT(  
    FLOAT _V  
) ;
```

Parameters

_V

The value to saturate.

Return value

The signed saturated value.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

[Functions](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_SRGB_to_FLOAT function

Article • 03/15/2021 • 2 minutes to read

Converts an SRGB value to FLOAT.

Syntax

syntax

```
FLOAT D3DX_SRGB_to_FLOAT(  
    UINT val  
)
```

Parameters

val

SRGB value to convert.

Return value

The converted SRGB value.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

[Functions](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_SRGB_to_FLOAT_inexact function

Article • 03/15/2021 • 2 minutes to read

Converts an SRGB value to FLOAT.

Syntax

syntax

```
FLOAT D3DX_SRGB_to_FLOAT_inexact(  
    hsls_precision FLOAT val  
) ;
```

Parameters

val

SRGB value to convert.

Return value

The converted SRGB value.

Remarks

This function doesn't have a high enough precision to give the exact answer. The alternative function [D3DX_SRGB_to_FLOAT](#) uses a lookup table to give an exact SRGB to float conversion.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

[Functions](#)

Unpacking and Packing DXGI_FORMAT for In-Place Image Editing

D3DX_Truncate_FLOAT function

Article • 03/15/2021 • 2 minutes to read

Truncates a FLOAT value.

Syntax

syntax

```
FLOAT D3DX_Truncate_FLOAT(  
    FLOAT _V  
) ;
```

Parameters

_V

The value to truncate.

Return value

The truncated value.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

[Functions](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_UINT2_to_R16G16_UINT function

Article • 03/15/2021 • 2 minutes to read

Packs the given XMUINT2 back into a DXGI_FORMAT_R16G16_UINT.

Syntax

syntax

```
UINT D3DX_UINT2_to_R16G16_UINT(
    hsls_precise XMUINT2 unpackedInput
);
```

Parameters

unpackedInput

The shader data to pack.

Return value

The packed shader data.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

Functions

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_UINT4_to_R10G10B10A2_UINT function

Article • 03/15/2021 • 2 minutes to read

Packs the given XMINT4 back into a DXGI_FORMAT_R10G10B10A2_UINT.

Syntax

syntax

```
UINT D3DX_UINT4_to_R10G10B10A2_UINT(  
    hsls_precision XMINT4 unpackedInput  
) ;
```

Parameters

unpackedInput

The shader data to pack.

Return value

The packed shader data.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

[Functions](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

D3DX_UINT4_to_R8G8B8A8_UINT function

Article • 03/15/2021 • 2 minutes to read

Packs the given XMUINT4 back into a DXGI_FORMAT_R8G8B8A8_UINT.

Syntax

syntax

```
UINT D3DX_UINT4_to_R8G8B8A8_UINT(  
    hsls_precise XMUINT4 unpackedInput  
) ;
```

Parameters

unpackedInput

The shader data to pack.

Return value

The packed shader data.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

Functions

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

Format conversion structures (HLSL reference)

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

This section contains the following structures:

- [XMINT2](#)
- [XMINT4](#)
- [XMUINT2](#)
- [XMUINT4](#)

Related topics

[Inline Format Conversion Reference](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

XMINT2 structure

Article • 05/26/2021 • 2 minutes to read

Describes an 2D integer vector.

Syntax

syntax

```
typedef struct _XMINT2 {  
    INT x;  
    INT y;  
} XMINT2;
```

Members

x

x-component of the vector.

y

y-component of the vector.

Remarks

This structure is defined in the `D3DX\DXGIFormatConvert.inl` header in the DirectX SDK (June 2010) for use from C++. The latest version of this header in the [Microsoft.DXSDK.D3DX](#) NuGet Package no longer defines it, and relies on `DirectX::XMINT2` in DirectXMath instead.

Requirements

Requirement	Value
Header	<code>D3DX_DXGIFormatConvert.inl</code>

See also

Structures

Unpacking and Packing DXGI_FORMAT for In-Place Image Editing

XMINT4 structure

Article • 05/26/2021 • 2 minutes to read

Describes an 4D integer vector.

Syntax

syntax

```
typedef struct _XMINT4 {
    INT x;
    INT {
        INT {
            INT w;
            }z;
        }y;
} XMINT4;
```

Members

x

x-component of the vector.

y

y-component of the vector.

z

z-component of the vector.

w

w-component of the vector.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

Remarks

This structure is defined in the `D3DX\DXGIFormatConvert.inl` header in the DirectX SDK (June 2010) for use from C++. The latest version of this header in the [Microsoft.DXSDK.D3DX](#) NuGet Package no longer defines it, and relies on `DirectX::XMINT4` in DirectXMath instead.

See also

[Structures](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

XMFLOAT2 structure

Article • 05/26/2021 • 2 minutes to read

Describes an 2D unsigned integer vector.

Syntax

syntax

```
typedef struct _XMUINT2 {  
    UINT x;  
    UINT y;  
} XMUINT2;
```

Members

x

x-component of the vector.

y

y-component of the vector.

Remarks

This structure is defined in the `D3DX\DXGIFormatConvert.inl` header in the DirectX SDK (June 2010) for use from C++. The latest version of this header in the [Microsoft.DXSDK.D3DX](#) NuGet Package no longer defines it, and relies on `DirectX::XMUINT2` in DirectXMath instead.

Requirements

Requirement	Value
Header	<code>D3DX_DXGIFormatConvert.inl</code>

See also

Structures

Unpacking and Packing DXGI_FORMAT for In-Place Image Editing

XMFLOAT4 structure

Article • 05/26/2021 • 2 minutes to read

Describes an 4D unsigned integer vector.

Syntax

syntax

```
typedef struct _XMUINT4 {  
    UINT x;  
    UINT {  
        UINT {  
            UINT w;  
            }z;  
        }y;  
    } XMUINT4;
```

Members

x

x-component of the vector.

y

y-component of the vector.

z

z-component of the vector.

w

w-component of the vector.

Remarks

This structure is defined in the `D3DX\DXGIFormatConvert.inl` header in the DirectX SDK (June 2010) for use from C++. The latest version of this header in the [Microsoft.DXSDK.D3DX](#) NuGet Package no longer defines it, and relies on [DirectX::XMUINT4](#) in DirectXMath instead.

Requirements

Requirement	Value
Header	D3DX_DXGIFormatConvert.inl

See also

[Structures](#)

[Unpacking and Packing DXGI_FORMAT for In-Place Image Editing](#)

Appendix (HLSL)

Article • 12/10/2020 • 2 minutes to read

The HLSL appendix lists additional information such as the words that are recognized as keywords by the language, and the grammar used for creating HLSL statements. It is published here for completeness.

- [Keywords \(DirectX HLSL\)](#)
- [Preprocessor Directives \(DirectX HLSL\)](#)
- [Reserved Words \(DirectX HLSL\)](#)
- [Grammar \(DirectX HLSL\)](#)

Related topics

[Reference for HLSL](#)

Keywords

Article • 06/30/2021 • 2 minutes to read

The Microsoft High Level Shader Language (HLSL) recognizes the words in this section as keywords. Keywords are predefined reserved identifiers that have special meanings. You can't use them as identifiers in your app.

- [AppendStructuredBuffer](#), `asm`, `asm_fragment`
- `BlendState`, `bool`, `break`, `Buffer`, `ByteAddressBuffer`
- `case`, `cbuffer`, `centroid`, `class`, `column_major`, `compile`, `compile_fragment`, `CompileShader`, `const`, `continue`, `ComputeShader`, `ConsumeStructuredBuffer`
- `default`, `DepthStencilState`, `DepthStencilView`, `discard`, `do`, `double`, `DomainShader`, `dword`
- `else`, `export`, `extern`
- `false`, `float`, `for`, `fxgroup`
- `GeometryShader`, `groupshared`
- `half`, `Hullshader`
- `if`, `in`, `inline`, `inout`, `InputPatch`, `int`, `interface`
- `line`, `lineadj`, `linear`, `LineStream`
- `matrix`, `min16float`, `min10float`, `min16int`, `min12int`, `min16uint`
- `namespace`, `nointerpolation`, `noperspective`, `NULL`
- `out`, `OutputPatch`
- `packoffset`, `pass`, `pixelfragment`, `PixelShader`, `point`, `PointStream`, `precise`
- `RasterizerState`, `RenderTargetView`, `return`, `register`, `row_major`, `RWBuffer`, `RWByteAddressBuffer`, `RWStructuredBuffer`, `RWTexture1D`, `RWTexture1DArray`, `RWTexture2D`, `RWTexture2DArray`, `RWTexture3D`
- `sample`, `sampler`, `SamplerState`, `SamplerComparisonState`, `shared`, `snorm`, `stateblock`, `stateblock_state`, `static`, `string`, `struct`, `switch`, `StructuredBuffer`
- `tbuffer`, `technique`, `technique10`, `technique11`, `texture`, `Texture1D`, `Texture1DArray`, `Texture2D`, `Texture2DArray`, `Texture2DMS`, `Texture2DMSArray`, `Texture3D`, `TextureCube`, `TextureCubeArray`, `true`, `typedef`, `triangle`, `triangleadj`, `TriangleStream`
- `uint`, `uniform`, `unorm`, `unsigned`
- `vector`, `vertexfragment`, `VertexShader`, `void`, `volatile`
- `while`

Remarks

These numeric types have scalar, vector, and matrix keyword expansions:

- `float`, `int`, `uint`, `bool`

- min10float, min16float
- min12int, min16int
- min16uint

The expansions of these numeric types follow this pattern, which uses float as an example:

- Scalar
float
- Vector
float1, float2, float3, float4
- Matrix
float1x1, float1x2, float1x3, float1x4 float2x1, float2x2, float2x3, float2x4 float3x1,
float3x2, float3x3, float3x4 float4x1, float4x2, float4x3, float4x4

HLSL supports lower-case [texture](#) and [sampler](#) for legacy reasons. Instead, for your new apps, we recommend that you use HLSL's new texture objects ([Texture2D](#), [Texture3D](#), and so on) and sampler objects ([SamplerState](#) and [SamplerComparisonState](#)).

export

Use **export** to mark functions that you package into a library.

Here is an example:

```
syntax

export float identity(float x)
{
    return x;
}
```

By marking the **identity** function with the **export** keyword, you make the **identity** function available from a library for later linking. Without the **export** marking, the **identity** function isn't available for later linking.

The compiler ignores the **export** keyword for non-library compilation.

Note

The **export** keyword requires the D3dcompiler_47.dll or a later version of the DLL.

Related topics

[Appendix \(DirectX HLSL\)](#)

Preprocessor Directives (HLSL)

Article • 06/02/2021 • 2 minutes to read

Preprocessor directives, such as `#define` and `#ifdef`, are typically used to make source programs easy to change and easy to compile in different execution environments.

Directives in the source file tell the preprocessor to perform specific actions. For example, the preprocessor can replace tokens in the text, insert the contents of other files into the source file, or suppress compilation of part of the file by removing sections of text. Preprocessor lines are recognized and carried out before macro expansion. Therefore, if a macro expands into something that looks like a preprocessor command, that command is not recognized by the preprocessor.

Preprocessor statements use the same character set as source file statements, with the exception that escape sequences are not supported. The character set used in preprocessor statements is the same as the execution character set. The preprocessor also recognizes negative character values.

The HLSL preprocessor recognizes the following directives:

- `#define`
- `#elif`
- `#else`
- `#endif`
- `#error`
- `#if`
- `#ifdef`
- `#ifndef`
- `#include`
- `#line`
- `#pragma`
- `#undef`

The number sign (#) must be the first nonwhite-space character on the line containing the directive; white-space characters can appear between the number sign and the first letter of the directive. Some directives include arguments or values. Any text that follows a directive (except an argument or value that is part of the directive) must be preceded by the single-line comment delimiter (//) or enclosed in comment delimiters /* */. Lines containing preprocessor directives can be continued by immediately preceding the end-of-line marker with a backslash (\).

Preprocessor directives can appear anywhere in a source file, but they apply only to the remainder of the source file.

Related topics

[Appendix \(DirectX HLSL\)](#)

#define directive (HLSL reference)

Article • 11/23/2019 • 2 minutes to read

Preprocessor directive that defines a constant or a macro.

Overload List

Item	Description
#define identifier token-string	Preprocessor directive that assigns a meaningful name to a constant in your application.
#define identifier(argument0, ..., argumentN-1) token-string	Preprocessor directive that creates a function-like macro.

See also

[Preprocessor Directives \(DirectX HLSL\)](#)

#define directive (constant)

Article • 11/23/2019 • 2 minutes to read

Preprocessor directive that assigns a meaningful name to a constant in your application.

```
#define identifier token-string
```

Parameters

Item	Description
<i>identifier</i>	Identifier of the constant.
<i>token-string</i> [optional]	Value of the constant. This parameter consists of a series of tokens, such as keywords, constants, or complete statements. One or more white-space characters must separate this parameter from the <i>identifier</i> parameter; this white space is not considered part of the substituted text, nor is any white space following the last token of the text. If you exclude this parameter, all instances of the <i>identifier</i> parameter are removed from the source file. The identifier remains defined, and can be tested using the #ifdef , #ifndef , and #ifndef directives.

Remarks

All instances of the *identifier* parameter that occur after the [#define](#) directive in the source file will be replaced by the value of the *token-string* parameter. The identifier is replaced only when it forms a token; for example, the identifier is not replaced if it appears in a comment, within a string, or as part of a longer identifier.

The [#undef](#) directive instructs the preprocessor to forget the definition of an identifier; for more information, see [#undef Directive \(DirectX HLSL\)](#).

Defining constants with the /D compiler option has the same effect as using the [#define](#) directive at the beginning of your file. Up to 30 constants can be defined with the /D option. For an example of how this can be used, see the Examples section of [#ifdef and](#)).

Examples

The following example defines the identifier WIDTH as the integer constant 80 and then defines LENGTH in terms of WIDTH and the integer constant 10.

```
#define WIDTH      80
#define LENGTH      ( WIDTH + 10 )
```

Every subsequent instance of LENGTH is replaced by (WIDTH + 10), and every subsequent instance of WIDTH + 10 is replaced by the expression (80 + 10). The parentheses around WIDTH + 10 are important because they control the interpretation in statements such as the following.

```
var = LENGTH * 20;
```

After the preprocessing stage the statement becomes the following, which evaluates to 1,800.

```
var = ( 80 + 10 ) * 20;
```

Without parentheses, the result would be the following, which evaluates to 280.

```
var = 80 + 10 * 20;
```

Related topics

[Preprocessor Directives \(DirectX HLSL\)](#)

[#define Overloads](#)

[#undef Directive \(DirectX HLSL\)](#)

#define directive (macro)

Article • 06/02/2021 • 2 minutes to read

Preprocessor directive that creates a function-like macro.

```
#define identifier( argument0, ... , argumentN-1 ) token-string
```

Parameters

Item	Description
<i>identifier</i>	Identifier of the macro. A second <code>#define</code> for a macro with an identifier that already exists in the current context generates an error unless the second token sequence is identical to the first.
(<i>argument0</i> , ... , <i>argumentN-1</i>)	List of arguments to the macro. The argument list is comma-delimited, can be of any length, and must be enclosed in parentheses. Each argument name in the list must be unique. No white space can separate the <i>identifier</i> parameter and the opening parenthesis. Use line concatenation place a backslash () immediately before the newline character to split long directives onto multiple source lines.
<i>token-string</i> [optional]	Value of the macro. This parameter consists of a series of tokens, such as keywords, constants, or complete statements. One or more white-space characters must separate this parameter from the <i>identifier</i> parameter; this white space is not considered part of the substituted text, nor is any white space following the last token of the text. Make liberal use of parentheses to ensure that complicated arguments are interpreted correctly. If the value of the <i>identifier</i> parameter occurs within the <i>token-string</i> parameter (even as a result of another macro expansion), it is not expanded. If you exclude this parameter, all instances of the <i>identifier</i> parameter are removed from the source file. The identifier remains defined, and can be tested using the <code>#if defined</code> , <code>#ifdef</code> , and <code>#ifndef</code> directives.

Remarks

All instances of the *identifier* parameter that occur after the `#define` directive in the source file constitute a macro call, and the identifier will be replaced by a version of the *token-string* parameter that has actual arguments substituted for formal parameters. The number of parameters in the call must match the number of parameters in the macro definition. The identifier is replaced only when it forms a token; for example, the

identifier is not replaced if it appears in a comment, within a string, or as part of a longer identifier.

The `#undef` directive instructs the preprocessor to forget the definition of an identifier; for more information, see [#undef Directive \(DirectX HLSL\)](#).

Defining constants with the `/D` compiler option has the same effect as using the `#define` directive at the beginning of your file. Up to 30 macros can be defined with the `/D` option.

The actual arguments in the macro call are matched to the corresponding formal arguments in the macro definition. Each formal argument in the token string is replaced by the corresponding actual argument, unless the argument is preceded by a stringizing (#), charizing (#@), or token-pasting (##) operator, or is followed by a ## operator. Any macros in the actual argument are expanded before the directive replaces the formal parameter.

Token pasting in the HLSL compiler is slightly different from token pasting in the C compiler, in that the pasted tokens must be valid tokens on their own. For example, consider the following macro definition:

```
#define MERGE(a, b) a##b  
MERGE(float, 4x4) test;
```

In the C compiler, this results in the following:

```
float4x4 test
```

In the HLSL compiler however, this results in the following:

```
float4 x4 test
```

You can work around this behavior by using the following macro definition instead.

```
MERGE(MERGE(float, 4), x4) test;
```

Examples

The following example uses a macro to define cursor lines.

```
#define CURSOR(top, bottom) (((top) << 8) | (bottom))
```

The following example defines a macro that retrieves a pseudorandom integer in the specified range.

```
#define getrandom(min, max) \  
((rand()%(int)((max) + 1)-(min))+(min))
```

Related topics

[Preprocessor Directives \(DirectX HLSL\)](#)

[#define Overloads](#)

[#undef Directive \(DirectX HLSL\)](#)

#error Directive

Article • 10/24/2019 • 2 minutes to read

Preprocessor directive that produces compiler-time error messages.

```
#error token-string
```

Parameters

Item	Description
<i>token-string</i>	Error message. This parameter consists of a series of tokens, such as keywords, constants, or complete statements. The token string is subject to macro expansion.

Remarks

#error directives are most useful for detecting programmer inconsistencies and violation of constraints during preprocessing. When an #error directive is encountered, compilation terminates.

Examples

The following example demonstrates error processing during preprocessing.

```
#if !defined(__cplusplus)
    #error C++ compiler required.
#endif
```

See also

[Preprocessor Directives \(DirectX HLSL\)](#)

#if, #elif, #else, and #endif Directives

Article • 08/19/2020 • 4 minutes to read

Preprocessor directives that control compilation of portions of a source file.

#if <i>ifCondition</i> ...
[#elif <i>elifCondition</i> ...]
[#else ...]
#endif

Parameters

Item	Description
<i>ifCondition</i>	Primary condition to evaluate. If this parameter evaluates to a nonzero value, all text between this #if directive and the next instance of the #elif, #else, or #endif directive is retained in the translation unit; otherwise, the subsequent source code is not retained. The condition can use the preprocessor operator defined to determine whether a specific preprocessor constant or macro is defined; this usage is equivalent to the use of the #ifdef directive. See the Remarks section for restrictions on the value of the <i>ifCondition</i> parameter.
<i>elifCondition</i> [optional]	Other condition to evaluate. If the <i>ifCondition</i> parameter and all previous #elif directives evaluate to zero, and this parameter evaluates to a nonzero value, all text between this #elif directive and the next instance of the #elif, #else, or #endif directive is retained in the translation unit; otherwise, the subsequent source code is not retained. The condition can use the preprocessor operator defined to determine whether a specific preprocessor constant or macro is defined; this usage is equivalent to the use of the #ifdef directive. See the Remarks section for restrictions on the value of the <i>elifCondition</i> parameter.

Remarks

Each #if directive in a source file must be matched by a closing #endif directive. Any number of #elif directives can appear between the #if and #endif directives, but at most one #else directive is allowed. The #else directive, if present, must be the last directive before #endif. Conditional-compilation directives contained in include files must satisfy the same conditions.

The #if, #elif, #else, and #endif directives can nest in the text portions of other #if directives. Each nested #else, #elif, or #endif directive belongs to the closest preceding #if directive.

If no conditions evaluate to a nonzero value, the preprocessor selects the text block after the #else directive. If the #else clause is omitted and no conditions evaluate to a nonzero value, no text block is selected.

The *ifCondition* and *elifCondition* parameters must meet the following requirements:

- Conditional compilation expressions are treated as [signed long](#) values, and these expressions are evaluated using the same rules as expressions in C++.
- Expressions must have integral type and can include only integer constants, character constants, and the defined operator.
- Expressions cannot use **sizeof** or a type-cast operator.
- The target environment may not be able to represent all ranges of integers.
- The translation represents type **int** the same as type **long**, and [unsigned int](#) the same as [unsigned long](#).
- The translator can translate character constants to a set of code values different from the set for the target environment. To determine the properties of the target environment, check values of macros from LIMITS.H in an application built for the target environment.
- The expression must not perform any environmental inquiries and must remain insulated from implementation details on the target computer.

Examples

This section contains examples that demonstrate how to use conditional compilation preprocessor directives.

Use of the defined operator

The following example shows the use of the defined operator. If the identifier CREDIT is defined, the call to the **credit** function is compiled. If the identifier DEBIT is defined, the call to the **debit** function is compiled. If neither identifier is defined, the call to the **printerror** function is compiled. Note that "CREDIT" and "credit" are distinct identifiers in C and C++ because their cases are different.

```
#if defined(CREDIT)
    credit();
```

```
#elif defined(DEBIT)
    debit();
#else
    printerror();
#endif
```

Use of nested #if directives

The following example shows how to nest #if directives. This example assumes that a symbolic constant named DLEVEL has been previously defined. The #elif and #else directives are used to make one of four choices, based on the value of DLEVEL. The constant STACK is set to 0, 100, or 200, depending on the definition of DLEVEL. If DLEVEL is greater than 5, then STACK is not defined.

```
#if DLEVEL > 5
    #define SIGNAL 1
    #if STACKUSE == 1
        #define STACK 200
    #else
        #define STACK 100
    #endif
#else
    #define SIGNAL 0
    #if STACKUSE == 1
        #define STACK 100
    #else
        #define STACK 50
    #endif
#endif
#if DLEVEL == 0
    #define STACK 0
#elif DLEVEL == 1
    #define STACK 100
#elif DLEVEL > 5
    display( debugptr );
#else
    #define STACK 200
#endif
```

Use for including header files

A common use for conditional compilation is to prevent multiple inclusions of the same header file. In C++, where classes are often defined in header files, conditional compilation constructs can be used to prevent multiple definitions. The following example determines whether the symbolic constant EXAMPLE_H is defined. If so, the file

has already been included and does not need to be reprocessed; if not, the constant EXAMPLE_H is defined to indicate that EXAMPLE.H has already been processed.

```
#if !defined( EXAMPLE_H )
#define EXAMPLE_H

class Example
{
...
};

#endif // !defined( EXAMPLE_H )
```

See also

[Preprocessor Directives \(DirectX HLSL\)](#)

#ifdef and #ifndef Directives

Article • 10/24/2019 • 2 minutes to read

Preprocessor directives that determine whether a specific preprocessor constant or macro is defined.

#ifdef <i>identifier</i> ...
#endif
#ifndef <i>identifier</i> ...
#endif

Parameters

Item	Description
<i>identifier</i>	Identifier of the constant or macro to check.

Remarks

You can use the #ifdef and #ifndef directives anywhere that the [#if](#) can be used. The #ifdef statement is equivalent to) directive. These directives check only for the presence or absence of identifiers defined using the [#define](#) directive, not for identifiers declared in the C or C++ source code.

These directives are provided only for compatibility with previous versions of the language. The use of the [defined](#) operator with the #if directive is preferred.

The #ifndef directive checks for the opposite of the condition checked by #ifdef. If the identifier is not defined, the condition is true (nonzero); otherwise, the condition is false (zero).

Examples

The identifier can be passed from the command line using the /D option. Up to 30 macros can be specified with /D. This is useful for checking whether a definition exists, because a definition can be passed from the command line. The following example uses this behavior to determine whether to run an application in test mode.

```
// PROG.CPP
#ifndef test
#define final
#endif
int main()
{
}
```

When compiled using the following command, prog.cpp will compile in test mode; otherwise, it will compile in final mode.

```
CL.EXE /Dtest prog.cpp
```

See also

[Preprocessor Directives \(DirectX HLSL\)](#)

[#if, \)](#)

#include Directive

Article • 08/20/2021 • 2 minutes to read

Preprocessor directive that inserts the contents of the specified file into the source program at the point where the directive appears.

```
#include "filename"
```

```
#include <filename>
```

Parameters

Item	Description
<i>filename</i>	Filename of the file to include, optionally preceded by a directory specification. The filename must specify an existing file.

Remarks

The #include directive causes replacement of the directive by the entire contents of the specified file. The preprocessor stops searching as soon as it finds a file with the specified name; if you specify a complete, unambiguous path specification for the file, the preprocessor searches only the specified path.

ⓘ Note

The **Effect-Compiler Tool** has a built-in include handler using the /I switch. However, when executing the compiler from the API, you can provide a customized include handler by implementing the ID3DXInclude interface.

The difference between the two syntax forms is the order in which the preprocessor searches for header files when the path is incompletely specified, as shown in the following table.

Syntax form	Preprocessor search pattern
-------------	-----------------------------

Syntax form	Preprocessor search pattern
#include "filename"	<p>Searches for the include file:</p> <ol style="list-style-type: none"> 1. in the same directory as the file that contains the #include directive. 2. in the directories of any files that contain a #include directive for the file that contains the #include directive. 3. in paths specified by the /I compiler option, in the order in which they are listed. 4. in paths specified by the INCLUDE environment variable, in the order in which they are listed. <p>[!NOTE] The INCLUDE environment variable is ignored in an development environment. Refer to your development environment's documentation for information about how to set the include paths for your project.</p>
#include <filename>	<p>Searches for the include file:</p> <ol style="list-style-type: none"> 1. in paths specified by the /I compiler option, in the order in which they are listed. 2. in paths specified by the INCLUDE environment variable, in the order in which they are listed. <p>[!NOTE] The INCLUDE environment variable is ignored in an development environment. Refer to your development environment's documentation for information about how to set the include paths for your project.</p>

Examples

The following example causes the preprocessor to replace the #include directive with the contents of stdio.h. Because the example uses the angle bracket format, the preprocessor will search for the file only in the directories listed by the /I compiler option and the INCLUDE environment variable.

C++

```
#include <stdio.h>
```

See also

- [Preprocessor Directives \(DirectX HLSL\)](#)
- [ID3D10Include Interface](#)
- [Effect-Compiler Tool](#)

#line Directive

Article • 10/24/2019 • 2 minutes to read

Preprocessor directive that sets the compiler's internally-stored line number and filename to the specified values.

```
#line lineNumber "filename"
```

Parameters

Item	Description
<i>lineNumber</i>	Line number to set. This can be any integer constant. Macro replacement can be performed on the preprocessing tokens, as long as the result evaluates to the correct syntax.
<i>filename</i> [optional]	Filename to set. The filename can be any combination of characters, and must be enclosed in double quotation marks (""). If this parameter is omitted, the previous filename remains unchanged.

Remarks

The compiler uses the line number and filename to refer to errors that it finds during compilation. The line number usually refers to the current input line, and the filename refers to the current input file. The line number is incremented after each line is processed. If you change the line number and filename, the compiler ignores the previous values and continues processing with the new values. The #line directive is typically used by program generators to cause error messages to refer to the original source file instead of to the generated program.

The translator uses the line number and filename to determine the values of the predefined macros `_FILE_` and `_LINE_`. You can use these macros to insert self-descriptive error messages into the program text. The `_FILE_` macro expands to a string whose contents are the filename, surrounded by double quotation marks ("").

Examples

The following example sets the line number to 151 and the filename to "copy.c".

```
#line 151 "copy.c"
```

In the following example, the macro ASSERT uses the predefined macros `_LINE_` and `_FILE_` to print an error message about the source file if the specified assertion is not true.

```
#define ASSERT(cond)

if( !(cond) )\
{printf( "assertion error line %d, file(%s)\n", \
_LINE_, _FILE_ );}
```

See also

[Preprocessor Directives \(DirectX HLSL\)](#)

#pragma Directive

Article • 10/24/2019 • 2 minutes to read

Preprocessor directive that provides machine-specific or operating system-specific features while retaining overall compatibility with the C and C++ languages.

```
#pragma token-string
```

Parameters

Item	Description
<i>token-string</i>	Series of characters that gives a specific compiler instruction and arguments. This parameter is subject to macro expansion.

Remarks

If the compiler finds a pragma it does not recognize, it issues a warning, but compilation continues.

The HLSL compiler recognizes the following pragmas:

- [def](#)
- [message](#)
- [pack_matrix](#)
- [warning](#)

See also

[Preprocessor Directives \(DirectX HLSL\)](#)

def pragma Directive

Article • 10/24/2019 • 2 minutes to read

Pragma directive that manually allocates a floating-point shader register.

```
#pragma def( target, register, val1, val2,val3, val4 )
```

Parameters

Item	Description
<i>target</i>	Target that contains the register to allocate.
<i>register</i>	Floating-point shader register to allocate.
<i>val0</i>	First byte of the value to allocate to the specified register.
<i>val1</i>	Second byte of the value to allocate to the specified register.
<i>val2</i>	Third byte of the value to allocate to the specified register.
<i>val3</i>	Fourth byte of the value to allocate to the specified register.

Remarks

The def pragma allows a developer to prefill a floating-point shader register with the specified value. This pragma is infrequently used.

See also

[Preprocessor Directives \(DirectX HLSL\)](#)

[#pragma Directive \(DirectX HLSL\)](#)

message pragma Directive

Article • 05/18/2021 • 2 minutes to read

Pragma directive that produces compiler-time messages.

```
#pragma message("token-string")
```

Parameters

Item	Description
<i>token-string</i>	Compiler message.

Examples

The following example demonstrates message processing during preprocessing.

```
#pragma message "Debugging flag set."
```

See also

[Preprocessor Directives \(DirectX HLSL\)](#)

[#pragma Directive \(DirectX HLSL\)](#)

pack_matrix pragma Directive

Article • 08/20/2021 • 2 minutes to read

Pragma directive that specifies packing alignment for matrices.

```
#pragma pack_matrix( alignment )
```

Parameters

Item	Description
<i>alignment</i>	Alignment to set for matrices. This parameter can take one of the values listed in the following table.
Value	Description
column_major	Default. Sets the matrix packing alignment to column major.
row_major	Sets the matrix packing alignment to row major.

Examples

The following example sets the matrix packing alignment to row major.

```
#pragma pack_matrix( row_major )
```

See also

[Preprocessor Directives \(DirectX HLSL\)](#)

[#pragma Directive \(DirectX HLSL\)](#)

warning pragma Directive

Article • 08/20/2021 • 2 minutes to read

Pragma directive that modifies the behavior of compiler warning messages.

```
#pragma warning( warning-specifier : warning-number-list [; warning-specifier : warning-number-list... ] )
```

Parameters

Item	Description
<i>warning-specifier</i>	Behavior to set for the specified warnings. This parameter can take one of the values listed in the following table.
Value	Description
once	Display the message of the warnings with the specified numbers only once.
default	Reset the behavior of the warnings with the specified numbers to their default value. This also has the effect of turning a warning on that is off by default. The warning will be generated at its default level.
1, 2, 3, 4	Apply the specified level to the warnings with the specified numbers. This also has the effect of turning a warning on that is off by default.
disable	Do not issue the warnings with the specified numbers.
error	Report the warnings with the specified numbers as errors.
<i>warning-number-list</i>	White space-delimited list of the numbers of the warnings to modify the behavior of.

Remarks

You can specify any number of distinct warning behavior changes within the same warning pragma by separating the changes with semicolons.

The compiler will add 4000 to any warning number that is between 0 and 999. For warning numbers greater than 4699, (those associated with code generation) the

warning pragma has effect only when placed outside function definitions. The pragma is ignored if it specifies a number greater than 4699 and is used inside a function.

The HLSL warning pragma does not support the **push** and **pop** functionality of the warning pragma included in the C++ compiler.

Examples

The following example disables warnings 4507 and 4034, displays warning 4385 once once, and reports warning 4164 as an error.

```
#pragma warning( disable : 4507 34; once : 4385; error : 164 )
```

The preceding example is functionally equivalent to the following:

```
#pragma warning( disable : 4507 34 )
#pragma warning( once : 4385 )
#pragma warning( error : 164 )
```

See also

[Preprocessor Directives \(DirectX HLSL\)](#)

[#pragma Directive \(DirectX HLSL\)](#)

#undef Directive

Article • 10/24/2019 • 2 minutes to read

Preprocessor directive that removes the current definition of a constant or macro that was previously defined using the [#define](#) directive.

```
#undef identifier
```

Parameters

Item	Description
<i>identifier</i>	Identifier of the constant or macro to remove the definition of. If you are undefining a macro, provide only the identifier, not the parameter list.

Remarks

You can apply the #undef directive to an identifier that has no previous definition; this ensures that the identifier is undefined. Macro replacement is not performed within #undef statements.

The #undef directive is typically paired with a [#define](#) directive to create a region in a source program in which an identifier has a special meaning. For example, a specific function of the source program can use manifest constants to define environment-specific values that do not affect the rest of the program. The #undef directive also works with the [] directive to control conditional compilation of the source program.

Constants and macros can be undefined from the command line using the /U option, followed by the identifiers to be undefined. This is equivalent to adding a sequence of #undef directives at the beginning of the source file.

Examples

The following example shows how to use the #undef directive to remove definitions of a symbolic constant and a macro.

```
#define WIDTH      80
#define ADD( X, Y ) (X) + (Y)
```

```
#undef WIDTH  
#undef ADD
```

See also

[Preprocessor Directives \(DirectX HLSL\)](#)

[#define Directive \(DirectX HLSL\)](#)

[#if, \)](#)

Reserved Words

Article • 06/30/2021 • 2 minutes to read

The following words are reserved for use by the HLSL language. Do not use them to name variables or functions in your HLSL code.

auto case, catch char, class, const_cast default, delete, dynamic_cast enum explicit friend
goto long mutable new operator private, protected, public reinterpret_cast short, signed,
sizeof, static_cast template, this, throw try, typename union, unsigned using virtual

Related topics

[Appendix \(DirectX HLSL\)](#)

Grammar

Article • 06/30/2021 • 2 minutes to read

HLSL statements are constructed using the following rules for grammar.

- [Whitespace](#)
- [Floating point numbers](#)
- [Integer numbers](#)
- [Characters](#)
- [Strings](#)
- [Identifiers](#)
- [Operators](#)
- [Related topics](#)

Whitespace

The following characters are recognized as white space.

- SPACE
- TAB
- EOL
- C style comments /* */
- C++ style comments //

Floating point numbers

Floating point numbers are represented in HLSL as follows:

- fractional-constant exponent-part(opt) floating-suffix(opt)
digit-sequence exponent-part floating-suffix(opt)
- fractional-constant :
digit-sequence(opt) . digit-sequence
digit-sequence .
- exponent-part :
e sign(opt) digit-sequence
E sign(opt) digit-sequence

- sign : one of
+ -
- digit-sequence :
digit
digit-sequence digit
- floating-suffix : one of
h H f F L

Use the “L” suffix to specify a full 64-bit precision floating-point literal. A 32-bit float literal is the default.

For example, the compiler recognizes the following literal value as a 32-bit precision floating-point literal and ignores the lower bits:

```
double x = -0.6473313946860445;
```

The compiler recognizes the following literal value as a 64-bit precision floating-point literal:

```
double x = -0.6473313946860445L;
```

Integer numbers

Integer numbers are represented in HLSL as follows:

- integer-constant integer-suffix(opt)
- integer-constant: one of
 - # (decimal number)
 - 0# (octal number)
 - 0x# (hex number)
- integer-suffix can be any one of these:

Characters

Characters are represented in HLSL as follows:

Character	Description
'c'	(character)
'\a' '\b' '\f' '\b' '\r' '\t' '\v'	(escapes)
'\###'	(octal escape, each # is an octal digit)
'\x#'	(hex escape, # is hex number, any number of digits)
'\c'	(c is other character, including backslash and quotation marks)

Escapes are not supported in preprocessor expressions.

Strings

Strings are represented in HLSL as follows:

"s" (s is any string with escapes).

Identifiers

Identifiers are represented in HLSL as follows:

```
[A-Za-z_][A-Za-z0-9_]*
```

Operators

```
##, #@, ++, --, &, &, &, ||, ==, ::, <<, <<=, >>, >>=, ...,
<=, >=, !=, *=, /=, +=, -=, %=, &=, |=, ^=, ->
```

Also any other single character that did not match another rule.

Related topics

[Appendix \(DirectX HLSL\)](#)

HLSL errors and warnings

Article • 03/15/2021 • 31 minutes to read

Error and warning codes that a shader can return.

Constant/value	Description
ERR_COMMENTEOF 1001	A comment continues past the end of file.
ERR_HEXTRUNCATED 1002	A hex value was truncated to 32 bits.
ERR_OCTTRUNCATED 1003	An octal value was truncated to 32 bits.
ERR_DECTRUNCATED 1004	A decimal value was truncated to 32 bits.
ERR_STRINGEOL 1005	A string continues past the end of line.
ERR_STRINGEOF 1006	A string continues past the end of file.
ERR_CHAR_EOF 1007	A character continues past the end of file.
ERR_TOK_VERSION 1008	An error in the token version.
ERR_PP_SYNTAX 1500	An invalid preprocessor syntax.
ERR_UNEXPECTEDTOKENS 1501	There were unexpected tokens following the preprocessor directive.
ERR_UNEXPECTEDEOF 1502	The end of file was reached unexpectedly.
ERR_DIVZERO 1503	A division by zero in the preprocessor expression occurred.
ERR_INVALIDCOMMAND 1504	An invalid preprocessor command.
ERR_INCLUDEFROMFILE 1505	The include interface that is required to support #include from resource or memory doesn't work.

Constant/value	Description
ERR_TOOMANYINCLUDES 1506	There are too many nested #includes.
ERR_FILE_OPEN 1507	The specified source file failed to open.
ERR_ELIF 1508	An unexpected #elif directive occurred.
ERR_ELSE 1509	An unexpected #else directive occurred.
ERR_ENDIF 1510	An unexpected #endif directive occurred.
ERR_DUPLICATEPARAMATER 1511	A duplicate parameter was supplied to the specified macro.
ERR_RESOURCE_OPEN 1512	A resource failed to open.
ERR_ELIF_ELSE 1513	An unexpected #elif directive followed a #else directive.
ERR_ELSE_ELSE 1514	An unexpected #else directive followed a #else directive.
ERR_UNEXPECTEDEOF_MACRO 1515	An unexpected end of file occurred in a macro expansion.
ERR_PARAMETERS_MACRO 1516	Not enough actual parameters were supplied to the specified macro.
ERR_PP_NOT_YET_IMPLEMENTED 1517	Functional defines in preprocessor expressions are not yet implemented.
ERR_INVALID_INT_EXPR 1518	An integer constant expression is invalid or unsupported.
ERR_MACRO_REDEFINITION 1519	The specified macro requires redefining.
ERR_LATE_FULL_PATH 1520	The #hsl_full_path directive must be the first content in a source file.
ERR_INVALID_FULL_PATH 1521	The #hsl_full_path directive was malformed.
ERR_PARSE_SYNTAX 3000	A syntax error was found while parsing a shader file.

Constant/value	Description
ERR_REDEFINITION 3003	The specified function requires redefining.
ERR_UNDECLARED_IDENTIFIER 3004	An undeclared identifier was found while parsing a shader file.
ERR_INVALID_USE 3005	The invalid use of a type was found while parsing a shader file.
ERR_EXTERN 3006	The specified variable can't be declared extern.
ERR_STATIC 3007	The specified variable can't be declared static.
ERR_VOLATILE 3008	The specified variable can't be declared volatile.
ERR_INITIALIZERS 3009	The specified variable can't have initializers.
ERR_GROUPSHARED 3010	The specified variable can't be declared as group shared or the group-shared variable can't perform a specific task.
ERR_NONLITERAL_INITIALIZER 3011	The specified variable must be a literal expression.
ERR_MISSING_INITIALIZERS 3012	The specified variable is missing initializers.
ERR_ARGUMENTS 3013	The function doesn't take the specified number of parameters.
ERR_ARGUMENTS_BASETYPE 3014	An incorrect number of arguments was passed to the numeric-type constructor.
ERR_ARGUMENTS_INTRINSIC 3015	An incorrect number of arguments was passed to the intrinsic function.
ERR_UNSUPPORTED_CAST 3017	The conversion from one type to another type is unsupported.
ERR_SUBSCRIPT 3018	The subscript is invalid.
ERR_NUMERIC_EXPECTED 3019	A numeric value, like a float, was expected.

Constant/value	Description
ERR_TYPE_MISMATCH 3020	A type mismatch occurred. For example, this error is returned if all template type components must have the same type, but they don't.
ERR_PARSE_ARRAY_EXPECTED 3021	An array was expected.
ERR_BASETYPE_EXPECTED 3022	A scalar, vector, or matrix was expected.
ERR_DETERMINANT 3023	A determinant error, such as a faulty observation, occurred.
ERR_VECTOR_EXPECTED 3024	A vector was expected.
ERR_LVALUE_EXPECTED 3025	An l-value, which specifies a const object, was expected.
ERR_MATRIX_MULTIPLICATION 3026	An error in matrix multiplication occurred.
ERR_PARSE_ARRAY_INDEX_OUT_OF_BOUNDS 3030	An index for an array is out of bounds.
ERR_PARSE_IMAGINARY_SQUARE_ROOT 3031	A imaginary square root error was found while parsing a shader file.
ERR_PARSE_INDEFINITE_LOG 3032	A indefinite log error was found while parsing a shader file.
ERR_PARSE_DIVISION_BY_ZERO 3033	A division-by-zero error was found while parsing a shader file.
ERR_CONST 3035	The specified variable can't be declared const.
ERR_REDEFINITION_FORMAL_PARAMETER 3036	An error occurred with the redefinition of the specified formal parameter.
ERR_UNSUPPORTED_TYPE_EXPR 3037	Constructors only defined for numeric base types.
ERR_NUMERIC 3038	The specified variable must be numeric.
ERR_PARSE_VERSION 3039	Can't be specific to the target.

Constant/value	Description
ERR_ANNOTATIONS 3040	Can't have annotations.
ERR_SHADER_VERSION 3041	The compiler target is unsupported.
ERR_PARSE_NOT_YET_IMPLEMENTED 3042	A not-yet-implemented error was found while parsing a shader file.
ERR_SEMANTICS 3043	Can't have semantics.
ERR_MISSING_DEFAULT_PARAMETER 3044	A default value for the specified parameter is missing.
ERR_OUTPUT_INITIALIZER 3045	Output only and can't be initialized.
ERR_CONST_OUTPUT 3046	Output parameters can't be declared const.
ERR_UNIFORM 3047	The specified variable can't be declared uniform.
ERR_USAGE 3048	Duplicate usages are specified.
ERR_USAGE_VERSION 3049	Can't be specific to the usage.
ERR_MATRIX_EXPECTED 3050	A matrix was expected.
ERR_SCALAR_EXPECTED 3051	A scalar was expected.
ERR_VECTOR_SIZE 3052	The vector dimension must be between 1 and MAX_VECTOR_SIZE.
ERR_MATRIX_SIZE 3053	The matrix dimensions must be between 1 and MAX_VECTOR_SIZE.
ERR_SHARED 3054	The specified variable can't be declared as shared.
ERR_INLINE 3055	The specified variable can't be declared as inline.
ERR_LITERAL_VARIABLE 3057	The specified variable is a literal expression.

Constant/value	Description
ERR_ARRAY_LITERAL 3058	Array dimensions must be literal scalar expressions.
ERR_ARRAY_SIZE 3059	Array dimension must be between 1 and 65536.
ERR_VECTOR_LITERAL 3060	The vector dimension must be a literal scalar expression.
ERR_MATRIX_LITERAL 3061	Matrix dimensions must be a literal scalar expressions.
ERR_UNIFORM_OUT 3062	The specified variable can't be declared 'uniform out'.
ERR_SAMPLER 3063	The specified variable can't be a sampler.
ERR_OBJECT_LITERALS 3064	Object literal expressions aren't allowed inside functions.
ERR_OBJECT_ASSIGNMENTS 3065	Object assignments aren't allowed inside functions.
ERR_SAMPLER_EXPECTED 3066	A sampler was expected.
ERR_AMBIGUOUS_FUNCTION_CALL 3067	The function call is ambiguous.
ERR_PROTOTYPE 3068	The return value of a function differs from the return value of the prototype of the function.
ERR_FUNCTION_HAS_BODY 3069	The function already has a body.
ERR_PARSE_INDEFINITE_ACOS 3070	A syntax error was found while parsing an indefinite arccosine.
ERR_PARSE_INDEFINITE_ASIN 3071	A syntax error was found while parsing an indefinite arcsine.
ERR_ARRAY_IMPLICIT 3072	Array dimensions for this type must be explicit.
ERR_ARRAY_IMPLICIT_ORDER 3073	Secondary array dimensions must be explicit.

Constant/value	Description
ERR_ARRAY_IMPLICIT_VALUE 3074	The implicit array is missing a value.
ERR_ARRAY_IMPLICIT_SIZE 3075	The implicit array type does not match.
ERR_VOID_SEMANTIC 3076	A void function can't have a semantic attached to it.
ERR_USAGE_MATRIX 3077	Non-matrix types can't be declared as row_major or column_major.
ERR_REDEFINITION_LOOP_CONTROL 3078	The loop control variable that is used outside the for-loop scope conflicts with a previous declaration in the outer scope; the most recent definition was used.
ERR_RETURN VOID 3079	Void functions can't return a value.
ERR_RETURN_VALUE 3080	The function must return a value.
ERR_COMMA_EXPRESSION 3081	A comma expression was used where an initializer list may have been intended.
ERR_BINARYTYPE_EXPECTED 3082	An int or unsigned int type is required for bitwise operators.
ERR_GEOMETRY_CONFLICT 3083	There are conflicting geometry types.
ERR_ATTRIBUTE 3084	Error with the attribute due to errors with its parameters.
ERR_UNSIGNED_TYPE 3085	The unsigned type can't be used with this variable type.
ERR_DEPRECATED_IN_STRICT_MODE 3086	The particular syntax (DirectX 9 syntax) or keyword (pixelshader) is deprecated in strict mode.
ERR_NO_METHODS 3087	The object doesn't have methods.
ERR_UNKNOWN_METHOD 3088	The object doesn't have the specified method.
ERR_TARGETUSAGE_INVALID 3089	The shader target or usage is invalid.

Constant/value	Description
ERR_NO_OBJECTS_IN_STRUCTS 3090	No writable textures, samplers, or UAVs can be members of compound types with interface inheritance.
ERR_PACK_OFFSET_IN_INVALID_SCOPE 3091	Packoffset is only allowed in a constant buffer.
ERR_UNARY_NEGATE_OF_UNSIGNED 3092	Unary negate of unsigned value is still unsigned.
ERR_OUT_OF_MEMORY 3093	Ran out of memory will performing the operation.
ERR_NON_COMPOUND_BASE 3094	The base type is not a structure, class, or interface.
ERR_MULTI_CONCRETE_BASE 3095	Multiple concrete base types were specified.
ERR_NOT_TEMPLATE_TYPE 3096	The specified variable isn't a template type.
ERR_STATIC_METHOD_MEMBER_USE 3097	The specified static method can't refer to instance members.
ERR_NO_METHOD_PROTOYPE 3098	The method isn't found in the class.
ERR_STATIC_METHOD_INSTANCE_CALL 3099	The specified static method can't be called on objects.
ERR_NO_STATIC_MEMBER_DECL 3100	The specified static member isn't found in the class or isn't a static variable.
ERR_STATIC_MEMBER_TYPE_MISMATCH 3101	The declaration type differs from the definition type.
ERR_INVALID_STATIC_MEMBER_SCOPE 3102	Static members can only be defined in global scopes.
ERR_MISSING_VARIABLE_DEFINITION 3103	The specified variable was declared but not defined.
ERR_NO_DATA_IN_INTERFACES 3104	Interfaces can't contain data.
ERR_NO_STATIC_METHODS_IN_INTERFACES 3105	Interface methods can't be static.

Constant/value	Description
ERR_NO_INTERFACE_METHOD_BODIES 3106	Interface methods can't be declared outside of an interface.
ERR_NO_INTERFACE_INHERITANCE 3107	Interfaces can't inherit from other types.
ERR_CLASS_MISSING_INTERFACE_METHOD 3108	The class doesn't implement the specified method.
ERR_MISMATCHED_OVERRIDE_RETTYPE 3109	The return type doesn't match the overridden method.
ERR_NO_INTERFACES_AS_MEMBERS 3110	Interfaces can't be members.
ERR_RECURSIVE_CONTAINMENT 3111	Types can't contain members of their own type.
ERR_NO_SWITCH 3112	Can't use call or forcecase attributes on switch statements in the specified programs.
ERR_NO_OUT_DEFAULTS 3113	
ERR_DEFAULT_REDEFINED 3114	Default parameters can only be provided in the first prototype.
ERR_CONFLICTING_REGISTER_SEMANTICS 3115	The specified register is used more than once.
ERR_INVALID_API_CALL 3116	The API call is invalid.
ERR_INVALID_COMPILE_EXPR_FLAG 3117	The debug info flag can only be set globally.
ERR_INTERFACE_OUT 3118	Interfaces can only be inputs.
ERR_MULTI_DIM_POINTER_ARRAY 3119	Interface arrays can't be multi-dimensional.
ERR_INVALID_INDEX 3120	Invalid type for index was specified. Index must be a scalar or a vector with the correct number of dimensions.
ERR_INDEXABLE_TYPE_EXPECTED 3121	An array, matrix, vector, or indexable object type was expected in the index expression.

Constant/value	Description
ERR_NON_SCALAR_VECTOR_ELEMENT 3122	The vector element type must be a scalar type.
ERR_NON_SCALAR_MATRIX_ELEMENT 3123	The matrix element type must be a scalar type.
ERR_OBJECT_HAS_OBJECT_ELEMENT 3124	The object element type can't be an object type.
ERR_INVALID_DOT_MIPS_USAGE 3125	The .mips type can only be used in a two-element indexing expression, such as, .mips[mip][element].
ERR_METHOD_IMPL_PROTO_MISMATCH 3126	The specified method didn't match any prototype in the class.
ERR_METHOD_IMPL_BODY_MISSING 3127	The specified method can't be re-declared.
ERR_NON_SIMPLE_STREAM 3128	Stream parameters can only be single-element types.
ERR_WARNING_AS_ERROR 3129	A warning was treated as an error.
ERR_FX_SINGLE 3130	The specified variable can't be declared 'single'.
ERR_NO_STATIC_INTERFACE_INIT 3131	Static interfaces can't have initializers.
ERR_NO_INTERFACES_IN_BUFFERS 3132	Interfaces can't be declared in buffers.
WAR_TYPE_MISMATCH 3200	Type mismatches aren't recommended.
WAR_NOFRAGMENTS 3201	Fragments aren't recommended.
WAR_INVALID_SEMANTIC 3202	The semantic doesn't apply and is ignored.
WAR_SIGNED_UNSIGNED_COMPARE 3203	A signed versus unsigned mismatch occurred between destination and value and unsigned is assumed.
WAR_INT_TOO_LARGE 3204	Unsigned integer literal is too large so is truncated.

Constant/value	Description
WAR_PRECISION_LOSS 3205	In the conversion from larger type to smaller, a loss of data might occur.
WAR_elt_TRUNCATION 3206	The implicit truncation of a vector type occurred.
WAR_CONST_INITIALIZER 3207	Initializer was used on a global 'const' variable. This requires setting an external constant. If a literal is wanted, use 'static const' instead.
WAR_FAILED_COMPILING_10L9VS 3208	Failed compiling the 10_level_9 (9_x feature levels) vertex shader version of the library function.
WAR_FAILED_COMPILING_10L9PS 3209	Failed compiling the 10_level_9 (9_x feature levels) pixel shader version of the library function.
ERR_COMP_NOT_YET_IMPLEMENTED 3500	The particular expressions are not yet implemented.
ERR_ENTRYPOINT_NOT_FOUND 3501	The entry-point function is not found.
ERR_MISSING_INPUT_SEMANTICS 3502	The specified input parameter is missing semantics.
ERR_MISSING_OUTPUT_SEMANTICS 3503	The specified output parameter or function return value is missing semantics.
ERR_COMP_ARRAY_INDEX_OUT_OF_BOUNDS 3504	The index of the array is out of bounds.
ERR_OLD_VERSION 3505	The version being used is no longer supported; instead use a current version.
ERR_UNRECOGNIZED_VERSION 3506	The compiler target isn't recognized.
ERR_RETURN 3507	The type can't return a value.
ERR_OUT_UNINITIALIZED 3508	The output parameter or return value was never assigned a value.

Constant/value	Description
ERR_DEPENDENT_TEX1D 3509	Texture sample is considered dependent since texcoord wasn't declared as at least a float.
ERR_FUNCTION_MISSING_BODY 3510	The function is missing an implementation.
ERR_CANT_UNROLL 3511	The loop is unable to unroll, the loop doesn't appear to terminate in a timely manner (in the specified number of iterations), or the unrolled loop is too large. Use the [unroll(n)] attribute to force an exact higher number.
ERR_ARRAY_INDEX_MUST_BE_LITERAL 3512	The index of the sampler array must be a literal expression.
ERR_COMP_ARRAY_EXPECTED 3513	An array or a particular array dimension was expected.
ERR_GEOMETRY_INVALID 3514	The specified input semantic is invalid for geometry shader primitives, it must be its own parameter.
ERR_TARGET_INVALID 3515	The target is invalid. For example, user-defined buffers can't be target specific, and the register specification expected a particular binding.
ERR_TEXCUBE_OFFSET_INVALID 3516	Texcube instructions can't have integer offsets.
ERR_UNDEFINED_VARIABLE 3517	The variable is undefined.
ERR_BREAK_OUTSIDE_LOOP 3518	A break must be inside a loop.
ERR_CONTINUE_OUTSIDE_LOOP 3519	A continue must be inside a loop.
ERR_TEXPROJ_INVALID_TEXCOORD 3520	Texture projection can't have texcoord instructions.
ERR_TEXTURE_TYPE 3521	The return type of the texture is too large. It can't exceed four components.
ERR_TEXTURE_OBJECTS_UNSUPPORTED 3522	Texture objects or streams aren't supported on legacy targets.

Constant/value	Description
ERR_COMPAT_MAKETEXTURE 3523	DirectX 9-style intrinsic functions are disabled when not running in DirectX 9 compatibility mode.
ERR_DUPLICATE_ATTRIBUTE 3524	Specific attributes can't be used together, like loop and unroll, or a duplicate attribute was supplied.
ERR_NOT_SIMPLE_LOOP 3525	The loop can't be mapped to a shader target because the target doesn't support breaks.
ERR_GRADIENT_WITH_BREAK 3526	Gradient instructions can't be used in loops with breaks.
ERR_TEXTURE_OFFSET 3527	Texture access requires literal offset and multisample index.
ERR_CANT_BRANCH 3528	Flow control (branching) can't be used on this profile.
ERR_MUST_BRANCH 3529	Flattening with flow control in this specific situation can't be done.
ERR_BIND_INVALID 3530	Invalid binding operation was performed. For example, buffers can only be bound to one slot or one constant offset; invalid register specification because a particular binding was expected but didn't occur; can't mix packoffset elements with nonpackoffset elements in a cbuffer.
ERR_NEED_UNROLL_FORCED_LOOP 3531	Loops that are marked with the loop attribute can't be unrolled.
ERR_DUPLICATE_CASE 3532	A duplicate default or case statement occurred in a switch statement.
ERR_MUST_HAVE_BREAK 3533	Non-empty case statements must have a break or return.
ERR_LOW_PRECISION 3534	Partial precision isn't supported for the specified target. Min-precision types might offer similar functionality.

Constant/value	Description
ERR_UNSUPPORTED_OPERATION 3535	An unsupported operation was performed. For example, bitwise operations aren't supported on legacy targets; CheckAccessFullyMapped requires shader model 5 or higher; TextureXxx methods for tiled resources require shader model 5 or higher.
ERR_INCOMPATIBLE_DUP_SEMANTICS 3536	SV_ClipDistance semantics can't be used when using the <code>clipplanes</code> attribute, or duplicated input semantics can't change type, size, or layout.
ERR_NO_FALLTHROUGH 3537	Fall-through cases in switch statements aren't supported. case/default statements that fall through to the next case/default without a break can't have any code in them.
ERR_NON_LITERAL_SAMPLER 3538	Sampler parameter must come from a literal expression.
ERR_OLDVERSION 3539	A particular shader version, such as, <code>ps_1_x</code> , is no longer supported; use <code>/Gec</code> in the <code>fxc.exe</code> HLSL code compiler to automatically upgrade to the next shader version, such as, <code>ps_2_0</code> ; alternately, <code>fxc</code> 's <code>/LD</code> option allows use of a previous compiler DLL.
ERR_NO_GLOBAL_PACK_OFFSETS 3540	Global packoffset variables aren't supported.
ERR_INVALID_PACK_OFFSET_NAME 3541	Invalid packoffset location was specified.
ERR_PACK_OFFSET_CANT_HAVE_TARGET 3542	A packoffset variable can't have a target qualifier.
ERR_REINTERPRET_UNSUPPORTED 3543	The operation can't reinterpret the supplied datatype.
ERR_NO_INTERFACE_SUPPORT 3544	Abstract interfaces aren't supported on the specified target; interface references must resolve to specific instances.
ERR_NO_IFACE_METHOD_IMPLS 3545	No classes implement the specified method.

Constant/value	Description
ERR_TBUFFER_UNSUPPORTED 3546	Reading from texture buffers is unsupported on the specified target.
ERR_NO_GLOBAL_COMPOUND_WRITES 3547	Global structs and classes can't be changed.
ERR_NO_NEGATIVE_EMULATED_UINTS 3548	The specified uints can only be used with known-positive values, use int if possible.
ERR_INTERLOCKED_TARGET 3549	Interlocked targets must be groupshared or UAV elements. Or, the specified target doesn't support interlocked operations, for example, IncrementCounter / DecrementCounter are only valid on RWStructuredBuffer objects.
WAR_ARRAY_INDEX_MUST_BE_LITERAL 3550	The index of the sampler array must be a literal expression, so the loop is forced to unroll.
WAR_INFINITE_LOOP 3551	An infinite loop was detected so the loop writes no values.
WAR_NOT_SIMPLE_LOOP 3552	The loop can't be mapped to a shader target because the target doesn't support breaks.
WAR_GRADIENT_WITH_BREAK 3553	Can't use gradient instructions in loops with break.
WAR_UNKNOWN_ATTRIBUTE 3554	The attribute is unknown or invalid for the specified statement.
WAR_INCOMPATIBLE_FLAGS 3555	Flags aren't compatible with the operation.
WAR_INT_DIVIDE_SLOW 3556	Integer divides might be much slower, try using uints if possible.
WAR_TOO_SIMPLE_LOOP 3557	The loop only executes for a limited number of iterations or doesn't seem to do anything so consider removing it or forcing it to unroll.
WAR_ENDIF_UNINITIALIZED 3558	The #endif directive is uninitialized.

Constant/value	Description
WAR_LOOP_ASYMMETRIC_RETURN 3559	The loop returns asymmetrically.
WAR_MUST_BRANCH 3560	If statements that contain out of bounds array accesses can't be flattened.
WAR_OLDVERSION 3561	A particular shader version, such as, ps_1_x, is no longer supported; use the next shader version, such as, ps_2_0.
WAR_OUTOFCOMMANDS_LOOPSIM 3562	The loop simulation goes out of bounds.
WAR_OUTOFCOMMANDS_LOOPUNROLL 3563	The loop unrolls out of bounds.
WAR_PRAGMA_RULEDISABLE 3564	For better compilation results, consider re-enabling the specified rule.
WAR_DID_NOT_SIMULATE 3565	Loop simulation finished early, use /O1 or higher for potentially better codegen.
WAR_NO_EARLY_BREAK 3566	Loop won't exit early, try to make sure the loop condition is as tight as possible.
WAR_IGNORED_REGISTER_SEMANTIC 3567	The register semantic is ignored.
WAR_UNKNOWN_PRAGMA 3568	The unknown pragma directive is ignored.
WAR_LOOP_TOO_LONG 3569	The loop executes for more than the maximum number of iterations for the specified shader target, which forces the loop to unroll.
WAR_GRADIENT_MUST_UNROLL 3570	A gradient instruction is used in a loop with varying iteration, which forces the loop to unroll.
WAR_POW_NOT_KNOWN_TO_BE_POSITIVE 3571	The <code>pow(f, e)</code> intrinsic function won't work for negative f, use <code>abs(f)</code> or conditionally handle negative values if you expect them.
WAR_VARYING_INTERFACE 3572	Interface references must resolve to non-varying objects.

Constant/value	Description
WAR_TESSFACTORSCALE_OUTOFRANGE 3573	Tessellation factor scale is clamped to the range [0, 1].
WAR_SYNC_IN_VARYING_FLOW 3574	Thread synchronization operations can't be used in varying flow control.
WAR_BREAK_FROM_UAV 3575	Automatic unrolling has been disabled for the loop, consider using the [unroll] attribute or manual unrolling. Or, loop termination conditions in varying flow control so can't depend on data read from a UAV.
WAR_OVERRIDDEN_SEMANTIC 3576	Patch semantics must live in the enclosed type so the outer semantic is ignored. Or, semantics in type are overridden by variable/function or enclosing type.
WAR_KNOWN_NON_SPECIAL 3577	The value can't be infinity, A call to isfinite might not be necessary. /Gis might force isfinite to be performed. Or, The value can't be NaN, A call to isnan might not be necessary. /Gis might force isnan to be performed.
WAR_TLOUT_UNINITIALIZED 3578	The output value isn't completely initialized.
WAR_GROUPSHARED_UNSUPPORTED 3579	The specified variable doesn't support groupshared so groupshared is ignored.
WAR_CONDITIONAL_SIDE_EFFECT 3580	Both sides of the &&, , or ?: operator are always evaluated so the side effect on the specified side won't be conditional.
WAR_NO_UNSIGNED_ABS 3581	The abs operation on unsigned values is not meaningful so it's ignored.
WAR_TEXTURE_OFFSET 3582	Texture access must have literal offset and multisample index.
WAR_POTENTIAL_RACE_CONDITION_UAV 3583	A race condition writing to a shared resource was detected, note that threads are writing the same value, but performance might be diminished due to contention.

Constant/value	Description
WAR_POTENTIAL_RACE_CONDITION_GSM 3584	A race condition writing to shared memory was detected, note that threads are writing the same value, but performance might be diminished due to contention.
WAR_UNRELIABLE_SOURCE_MARK 3585	Source_mark is most useful in /Od builds. Without /Od source_mark, can be moved around in the final shader by optimizations.
WAR_NO_INTERFACE_SUPPORT 3586	Abstract interfaces aren't supported on the specified target so interface references must resolve to specific instances.
WAR_MIN10_RCP 3587	The target emulates A / B with A * reciprocal(B). If the reciprocal of B is not representable in your min-precision type, the result might not be mathematically correct.
WAR_NO_CLIPPLANES_IN_LIBRARY 3588	The clipplanes attribute is ignored in library functions.
ERR_PRAGMA_DEF_OBSOLETE 3589	The '#pragma def' directive is no longer supported on DirectX 10+ and 10_level_9 (9_x feature levels) targets. Use compatibility mode to allow compilation.
ERR_NO_32_BIT_HALF 3650	Global variables can't use the 'half' type in the specified target. To treat this variable as a float, use the backwards compatibility flag.
ERR_NO_32_BIT_DOUBLE 3651	The specified target doesn't support double data type values.
ERR_NO_SMALL_INT 3652	The specified target doesn't support 8-bit or 16-bit integers.
ERR_NO_64_BIT_INT 3653	The specified target doesn't support 64-bit integers.
ERR_NO_UNSIGNED_ABS 3654	The abs operation on unsigned values isn't supported.
ERR_THREAD_GROUP_SIZE_INVALID 3655	The thread group size is invalid.

Constant/value	Description
ERR_THREAD_GROUP_SIZE_MISSING 3656	The size of the thread group is missing.
ERR_HSATTRIBUTE_INVALID 3657	Expected the specified parameter to be a certain value but got the specified value. Or, line or triangle output topologies are only available with isoline domains. Or, the maximum tessellation factor must be in the range [1,64].
ERR_HS_PATCH_INVALID 3658	Only one InputPatch or OutputPatch parameter is allowed. Or, InputPatch inputs can only be used in hull and geometry (5_0+) shaders. Or, OutputPatch inputs can only be used in the domain shaders and a hull shader's patch constant function.
ERR_HS_TYPE_MISMATCH 3659	The patch constant function must use the same input control point type that is declared in the control point phase. Or, the patch constant function must use the same output control point type that is returned from the control point phase. Or, the patch constant function's output patch input should have a certain number of elements, but has the specified amount.
ERR_INTERLOCKED_UNSUPPORTED 3660	The specified target doesn't support interlocked operations.
ERR_GROUPSHARED_UNSUPPORTED 3661	The specified variable doesn't support groupshared .
ERR_INDETERMINATE_DERIVATIVE 3662	The gradient operation uses a value that might not be defined for all pixels (in the specified target, UAV loads can't participate in gradient operations).
ERR_SYNC_IN_VARYING_FLOW 3663	Thread synchronization operations can't be used in varying flow control.
ERR_SYNC_UNSUPPORTED 3664	The specified target doesn't support synchronization operations.
ERR_NO_APPEND_CONSUME 3665	The specified target doesn't support Append/Consume buffers.

Constant/value	Description
ERR_NO_TYPED_UAVS 3666	The specified target doesn't support typed UAVs.
ERR_NO_UAVS 3667	The specified target doesn't support UAVs.
ERR_INDEX_IS_NOT_GROUP_INDEX 3668	Stores to group shared memory for specified targets must be indexed by an SV_GroupIndex only.
ERR_NON_LITERAL_RESOURCE 3669	Resources being indexed can't come from conditional expressions, they must come from literal expressions.
ERR_NON_LITERAL_STREAM 3670	The stream parameter must come from a literal expression.
ERR_BREAK_FROM_UAV 3671	Loop termination conditions in varying flow control so can't depend on data read from a UAV.
ERR_NO_PULL_MODEL 3672	The specified target doesn't support pull-model attribute evaluation.
ERR_CANT_PULL_POSITION 3673	The specified target doesn't support pull-model evaluation of position.
ERR_PULL_MUST_BE_INPUT 3674	Attribute evaluation can only be done on values that are taken directly from inputs.
ERR_LOOP_CONDITION_OUT_OF_BOUNDS 3675	Can't unroll loop with an out-of-bounds array reference in the condition.
ERR_TYPED_UAV_LOAD_MULTI_COMP 3676	Typed UAV loads are only supported for single-component 32-bit element types.
ERR_MULTIPLE_DEPTH_OUT 3677	The specified target only allows one depth output.
ERR_NO_ORDERED_ACCESS_IN_INTERFACE 3678	Interface-reachable members containing UAVs or group shared variables aren't implemented yet.
ERR_COMP_GLC_INVALID 3679	The storage class <code>globallycoherent</code> can only be used with Unordered Access View (UAV) buffers and can't be used with append/consume buffers.

Constant/value	Description
ERR_HS_UNKNOWN_OUTPUT_TYPE 3680	When you define a pass-through control-point shader, you must declare an InputPatch object, and the number of output control points must be zero or must match the input patch size.
ERR_ATOMIC_REQUIRES_INT 3681	The specified target only supports interlocked operations on scalar int or uint data.
ERR_ATTRIBUTE_PARAM_SIDE_EFFECT 3682	Expressions with side effects are invalid as attribute parameters
ERR_INVALID_RESOURCE_CONTAINER 3683	Groupshared variables can't contain resources such as textures, samplers or UAVs. Or, resources such as textures, samplers or UAVs can't contain other resources.
ERR_UNSUPPORTED_DOUBLE_OPERATION 3684	The specified target doesn't support double-precision floating-point. Or, the operation can't be used directly on resources. Or, the operation can't be used with doubles, cast to float first. Or, the operation isn't supported on the given type.
ERR_INVALID_TESS_FACTOR_SEMANTIC 3685	The tessfactor semantic is out of order. Or, conflicting quad/tri/isoline tessfactor semantic. Or, tessfactor semantics must be in the same component.
ERR_UNSUPPORTED_THIS_OBJECT 3686	The specified object isn't supported.
ERR_INVALID_SHADER_IO 3687	Double types can't be used as shader inputs or outputs. If you need to pass a double between shader stages, you must pass it as two uints and use asuint and asddouble to convert between forms.
ERR_INDEXED_DERIV 3688	Derivatives of indexed variables aren't implemented yet.
ERR_ORDERED_ACCESS_CAST 3689	The left-hand side of an assignment can't be cast to an indexable object so consider using asuint , asfloat , or asddouble on the right-hand side.

Constant/value	Description
ERR_RESOURCE_UNINITIALIZED 3690	The resource being indexed is uninitialized.
ERR_INVALID_STATIC_VAR_INIT 3691	Invalid variable reference in static variable initializer. Locals can't be used to initialize static variables.
ERR_NO_ABORT 3692	The specified target doesn't support aborts.
ERR_NO_MESSAGES 3693	The specified target doesn't support messages.
ERR_GUARANTEED_RACE_CONDITION_UAV 3694	A race condition writing to a shared resource was detected so consider making this operation write conditional.
ERR_GUARANTEED_RACE_CONDITION_GSM 3695	A race condition writing to shared memory was detected so consider making this operation write conditional.
ERR_INFINITE_LOOP 3696	An infinite loop was detected so the loop never exits.
ERR_TEMPLATE_VAR_CONFLICT 3697	The specified variable matches a variable in the template shader but the type layout doesn't match.
ERR_RESOURCE_BIND_CONFLICT 3698	The specified resource had binding conflicts with the template shader.
ERR_COMPLEX_TEMPLATE_RESOURCE 3699	Place-holder template resources can only be simple resources so structs and arrays aren't supported.
ERR_RESOURCE_NOT_IN_TEMPLATE 3700	For the specified resource, binding isn't present in the template shader.
ERR_RESINDEX_UNSUPPORTED 3701	The specified target doesn't support indexing resources.
ERR_FMA_ONLY_DOUBLE 3702	The fma intrinsic function can only be used with double arguments.
ERR_NO_MIN_PRECISION 3703	The specified target doesn't support minimum-precision data.
ERR_NO_F32_F16 3704	The specified target doesn't support 16-bit float conversions.

Constant/value	Description
ERR_NOT_ABLE_TO_FLATTEN 3705	If statements that contain side effects can't be flattened.
ERR_INVALID_MININT 3706	Signed integer division isn't supported on minimum-precision types. Cast to int to use 32-bit division.
ERR_INVALID_MIN8FLOAT 3707	A minimum 8-bit floating point value is invalid or unsupported.
ERR_CONTINUE_INSIDE_SWITCH 3708	A continue statement can't be used in a switch statement.
ERR_DEBUG_NOT_SUPPORTED_FOR_MODERN 3709	Debug isn't supported.
ERR_UNSUPPORTED_PARAM_TYPE 3710	The specified function parameters are unsupported.
ERR_DUPLICATE_FUNC_PARAM_SEMANTICS 3711	Library function parameters and return values can't have duplicate semantic.
ERR_LIBRARY_FUNC_UNSUPPORTED 3712	Library functions are supported only for pixel shaders and vertex shaders.
ERR_ENTRYPOINT_MUST_BE_EMPTY 3713	An entry point can't be specified for a library. Mark library entry points with the export keyword.
ERR_NO_STATIC_IN_LIBRARY 3714	The specified variable is declared as static, which isn't supported for libraries yet.
ERR_NO_TBUFFER_IN_LIBRARY 3715	The specified variable is declared as tbuffer, which is not supported for libraries yet.
ERR_NO_INTERFACES_IN_LIBRARY 3716	Classes and interfaces aren't supported in libraries.
ERR_NO_DOUBLE_IN_LIBRARY 3717	Double data types can't be used as library function inputs or outputs. If you need to pass a double to a library function, you must pass it as two uints and use asuint and asdouble to convert between forms.
ERR_NO_OVERLOADING_FOR_LIB_FUNC 3718	Library entry points can't be overloaded.

Constant/value	Description
ERR_RES_MAY_ALIAS_ONLY_IN_CS_5 3719	The 'resources_may_alias' option is only valid for cs_5_0+ targets.
ERR_READ_BEFORE_WRITE 4000	The specified variable is used without having been completely initialized.
ERR_MID_DIVISION_BY_ZERO 4001	A division by zero in the mid-level preprocessor expression occurred.
ERR_MID_INDEFINITE_LOG 4002	An indefinite logarithm occurred.
ERR_MID_IMAGINARY_SQUARE_ROOT 4003	An imaginary square root occurred.
ERR_TOO_COMPLEX 4004	The program is too complex because there are more active values than registers.
ERR_INDEFINITE_ASIN 4005	An indefinite arcsine occurred.
ERR_INDEFINITE_ACOS 4006	An indefinite arccosine occurred.
ERR_ARRAY_INDEX_OUT_OF_BOUNDS 4007	The array index is out of bounds.
WARN_FLOAT_DIVISION_BY_ZERO 4008	A floating point division by zero occurred.
ERR_IDIV_DIVISION_BY_ZERO 4009	An integer division by zero occurred.
ERR_UDIV_DIVISION_BY_ZERO 4010	An unsigned integer division by zero occurred.
ERR_FTOI_OUTOFRANGE 4011	The floating-point value out of integer range for a conversion.
ERR_FTOU_OUTOFRANGE 4012	The floating-point value out of unsigned integer range for a conversion.
ERR_INDEFINITE_DSXY 4013	An indefinite derivative calculation occurred.
ERR_GRADIENT_FLOW 4014	Gradient operations can't occur inside loops with divergent flow control.

Constant/value	Description
ERR_MID_SEMANTIC_TOO_LONG 4015	The semantic length is too long.
ERR_INVALID_SEMANTIC 4016	The semantic is invalid. For example, the <code>SV_InstanceID</code> semantic can't be used with <code>10_level_9</code> (<code>9_x</code> feature levels) targets, or zero-character semantics aren't supported.
ERR_MID_INVALID_REGISTER_SEMANTIC 4017	The same variable can't be bound to multiple constants in the same constant bank.
ERR_TOO_MANY_PHASES 4018	The shader uses texture addressing operations in a dependency chain that is too complex for the specific target shader model to handle.
ERR_CONSTANT_REG_COLLISION 4019	Multiple variables were found with the same user-specified location.
ERR_TBUFFER_REG_COLLISION 4020	Multiple variables were found with the same user-specified location.
ERR_DERIV_READ_BEFORE_WRITE 4021	Derivative is being used before it was defined so consider moving the derivative assignment earlier in the program.
ERR_DERIV_INVALID_PREDICATE 4022	Derivative isn't defined in a different branch of flow-control so consider moving the derivative assignment before any flow control statements.
ERR_DERIV_REDEFINITION 4023	A redefinition of a derivative occurred, and derivatives can only be assigned once.
ERR_DERIV_KNOWN_VALUE 4024	Derivatives of known values are unimplemented.
ERR_DERIV_UNKNOWN 4025	Unable to calculate the derivative of the specified value.
ERR_RACE_CONDITION_INDUCED_INV_SYNC 4026	A thread sync operation must be in non-varying flow control. Because of a potential race condition, this sync is invalid so consider adding a sync after reading any values that control shader execution at this point.

Constant/value	Description
ERR_ALIAS_ARRAY_INDEX_OUT_OF_BOUNDS 4027	The array index is out of bounds.
ERR_MINPRECISION_PRECISE 4028	The specified variable has a minimum precision type and can't be marked precise.
ERR_LOOP_NEVER_BREAKS 4029	An infinite loop was detected so the loop never exits.
WARN_FTOI_OUTOFRANGE 4114	The literal floating-point value is out of integer range for the conversion.
WARN_FTOU_OUTOFRANGE 4115	The literal floating-point value is out of unsigned integer range for the conversion.
WARN_IDIV_DIVISION_BY_ZERO 4116	A possible integer divide by zero occurred.
WARN_UDIV_DIVISION_BY_ZERO 4117	A possible unsigned integer divide by zero occurred.
WARN_IMAGINARY_SQUARE_ROOT 4118	An imaginary square root operation occurred.
WARN_INDEFINITE_LOG 4119	An indefinite logarithm operation occurred.
WARN_REPLACE_NOT_CONVERGE 4120	Optimizations aren't converging.
WARN_HOISTING_GRADIENT 4121	Gradient-based operations must be moved out of flow control to prevent divergence. Performance might improve by using a non-gradient operation.
WARN_FLOAT_PRECISION_LOSS 4122	The sum of two floating point values can't be represented accurately in double precision.
WARN_FLOAT_CLAMP 4123	Floating-point operations flush denorm float literals to zero so the specified floating point value is losing precision (this warning will only be shown once per compile).
ERR_GEN_NOT_YET_IMPLEMENTED 4500	A feature like clipping from a swizzled vector is not yet implemented.

Constant/value	Description
ERR_DUPLICATE_INPUT_SEMANTIC 4501	An inconsistent semantic definition occurred.
ERR_INVALID_INPUT_SEMANTIC 4502	The specified input semantic is invalid.
ERR_INVALID_OUTPUT_SEMANTIC 4503	The specified output semantic is invalid.
ERR_DUPLICATE_OUTPUT_SEMANTIC 4504	Overlapping output semantics occurred.
ERR_MAX_TEMP_EXCEEDED 4505	The maximum temp register index was exceeded.
ERR_MAX_INPUT_EXCEEDED 4506	The maximum number of inputs was exceeded.
ERR_MAX_CONST_EXCEEDED 4507	The maximum constant register index was exceeded. Try to reduce the number of constants that are referenced.
ERR_MAX_ADDR_EXCEEDED 4508	The maximum address register index was exceeded.
ERR_GEN_INVALID_REGISTER_SEMANTIC 4509	An invalid register semantic was used, or a variable must be bound to multiple register banks.
ERR_MAX_SAMPLER_EXCEEDED 4510	The maximum number of samplers was exceeded.
ERR_REL_ADDRESS_NOT_SUP 4511	The target doesn't support relative addressing.
ERR_NO_W_ACCESS 4512	The texture coordinate w-component can't be accessed.
ERR_NO_DEP_FROM_COL 4513	Dependent texture read operations that in any way are based on color inputs can't be performed.
ERR_PROGRAM_TOO_BIG 4514	The program is too big.
ERR_CANNOT_BIND_SAMPLER 4515	The sampler can't be bound to the user specified stage or sampler array.
ERR_CANNOT_READ SAME_TEX 4516	A texcoord that was used as input in a sampler can't be read from.

Constant/value	Description
ERR_CONFLICT_SAMP_BIND 4517	User defined sampler or sampler array bindings are conflicting. If two samplers have the same user binding, they can't both be used in the same shader.
ERR_MULTI_READ_SAMP_BIND 4518	Texture lookup can't be performed twice from a user bound or similar array access sampler.
ERR_TOO_MANY_TEXREADS 4519	Too many texture loads and reads occurred from texcoords.
ERR_NO_TEXCRD_SHARE 4520	texcoord can be read from and used for texlookup only in ps_1_4 and higher.
ERR_OUT_OF_TEMP 4521	The program is too complex and is out of temporary registers.
ERR_NO REP_SWIZZLE 4522	Replicate swizzles are only supported in ps_1_4.
ERR_NO_DEP_MATCH 4523	This dependent texture read can't be mapped to ps_1_x, or the shader can't compile to a ps_1_x shader because this model can't match all the dependent texture reads this shader requires.
ERR_TEXM_NO_SHARE 4524	texm can't be matched because computed texcoord is used in shader.
ERR_TEXM_NOT_COR_STAGE 4525	texm* can't be matched because source inputs aren't in the appropriate texture coordinates. For more info, see the ps_1_x assembly reference .
ERR_TEXM_NO_SOURCE_MOD 4526	texm* can't be matched to because texm* can't have source modifiers on input texcoord.
ERR_TEXM_BX2_ONLY 4527	texm* can't be matched to because texm* can only have bx2 modifier on input texload.
ERR_DEPTH_SCALAR 4528	DEPTH must be a scalar.
ERR_COLOR_4COMP 4529	The semantic (SV_Target or COLOR) value must be a four-component vector.

Constant/value	Description
ERR_WRITE_TO_COLOR0 4530	The pixel shader must minimally write all four components of the semantic (SV_Target0 or COLOR0) value.
ERR_DP4_NOT_SUP 4531	DP4 isn't supported.
ERR_NO_MATCH 4532	The expression can't be mapped to the shader instruction set.
ERR_NO_SWIZZLE_MATCH 4533	Swizzle can't be mapped to ps_1_x.
ERR_NO_DOUBLE_DEP 4534	Double dependent texture reads can't be performed in ps_1_x.
ERR_NO_TEX_SOURCE 4535	The texreg2ar or texreg2gb instruction can't be matched to because you can't have input modifiers.
ERR_TEXRGB_NOT_SUPPORTED 4536	The expression can only be mapped to texreg2rgb, but this instruction isn't supported on 1_x.
ERR_CANT_EMMULLATE_WRITE 4537	Write masks can't be emulated for the ps_1_x shader model.
ERR_COLOR_CONT 4538	SV_Target outputs must be contiguous from SV_Target0 to SV_TargetN, or COLOR outputs must be contiguous from COLOR0 to COLORn.
ERR_SAMPLER_MISMATCH 4539	A sampler mismatch occurred because the sampler was used inconsistently.
ERR_SEMANTIC_SCALER 4540	PSIZE or FOG must be a scalar.
ERR_WRITE_ALL_POS 4541	The vertex shader must minimally write all four components of SV_Position or POSITION.
ERR_TEXCOORD_CONT 4542	Texcoord outputs must be contiguous from texcoord0 to texcoordn.
ERR_NO_MULTI_SEM 4543	Multi-register semantics aren't supported in fragments.

Constant/value	Description
ERR_NO_4COMP_CLIP 4544	The clip must be from a 3 vector in ps_1_x.
ERR_TEXTURE_NOT_SUPPORTED 4545	An unsupported texture type for the specified target was encountered.
ERR_MAX_SAMP_EXCEEDED 4546	The maximum sampler register index was exceeded.
ERR_DEBUG_SIZE 4547	The debug info exceeds the maximum comment size so no debug info was emitted.
ERR_CONSTANTABLE_SIZE 4548	The constant table info exceeds the maximum comment size.
ERR_MAX_PRED_EXCEEDED 4549	The maximum predicate register index was exceeded.
ERR_MAX_BOOL_EXCEEDED 4550	Try reducing the number of constant branches, take bools out of structs/arrays, or move them to the start of the struct.
ERR_MAX_LOOP_EXCEEDED 4551	Try reducing the number of loops, take loop counters out of structs/arrays, or move them to the start of the struct.
ERR_NOT_SIMPLE_FOR 4552	The general loop can't be mapped to this instruction set.
ERR_ADDRESS_TOO_DEEP 4553	Relative address references are too deep.
ERR_CND_SCALAR 4554	Vector conditionals can't be emulated in ps_1_x shader model.
ERR_INVALID_TYPE 4555	An invalid type used for the specified semantics.
ERR_MAX_TEXTURE_EXCEEDED 4565	The maximum number of texture slots is exceeded for a library.
ERR_REQUIRE_INT_OFFSET 4566	Offset texture instructions must take an offset, which can resolve to integer literal in the range -8 to 7.
ERR_MAX_CBUFFER_EXCEEDED 4567	The maximum number of constant buffer slots is exceeded for a library.

Constant/value	Description
ERR_INCORRECT_USAGE 4568	The usage is unsupported on the target. For example, the sample interpolation, nointerpolation, noperspective, or integer inputs usages might be unsupported.
ERR_POSITION_INCORRECTTYPE 4569	An incorrect type was specified for the POSITION value.
ERR_MULTIPLE_STREAMS 4570	The target can only emit to a specific amount of streams.
ERR_MAX_OUTPUT_EXCEEDED 4571	The output limit was exceeded.
ERR_NO_STREAMS_USED 4572	The geometry shader didn't emit anything.
ERR_GEN_SEMANTIC_TOO_LONG 4573	The semantic length is too long and is limited to the specified number of characters.
ERR_DUPLICATE_SYSVAL_SEMANTIC 4574	A duplicate system value semantic definition was encountered.
ERR_READING_UNINITIALIZED 4575	An uninitialized value was read.
ERR_SIGNATURE_VALIDATION 4576	An error occurred during signature validation.
ERR_INCOMPLETE_POSITION 4577	Not all elements of SV_Position were written.
ERR_DUPLICATE_CBUFFER_BANK 4578	The specified cbuffer register was used more than once.
ERR_INVALID_FP_LITERAL 4579	An invalid floating point literal occurred.
ERR_UNWRITTEN_SI_VALUE 4580	The specified output contains a system- interpreted value that must be written in every execution path of the shader. Unconditional initialization might help.
ERR_AUTOSAMPLER_ARRAY_UNIMPL 4581	Using sampler arrays with texture objects on 10_level_9 (9_x feature level) targets isn't implemented yet.

Constant/value	Description
ERR_INVALID_TEXTURE_FORMAT 4582	Sampling from non-floating point texture formats can't be done.
ERR_INVALID_10L9_SEMANTIC 4583	The specified semantic isn't supported on the 10_level_9 (9_x feature level) target.
ERR_MAX_IFACE_EXCEEDED 4584	The maximum number of interface pointers was exceeded.
ERR_MAX_UAV_EXCEEDED 4585	The maximum number of UAV slots was exceeded for a library.
ERR_MAX_GROUP_SHARED_MEMORY_EXCEEDED 4586	The total amount, in bytes, of group shared memory exceeded the target's limit.
ERR_TOO_MANY_GROUP_SHARED_DATA 4587	Shaders compiled for the specified target can only have a single group shared data item.
ERR_INCORRECT_NUM_GROUP_SHARED_ELEMENTS 4588	Group shared data for the specified target must have a count of elements that is equal to the number of threads in the thread group.
ERR_CONTROL_POINT_COUNT_EXCEEDED 4589	The maximum control point count for the target was exceeded
ERR_GROUP_SHARED_DATA_ELEMENT_TOO_LARGE 4590	Group shared data for the specified target is too large and must have an element size of at most the specified amount of bytes when compiling for the specified number of threads.
ERR_GROUP_SHARED_DATA_NOT_AN_ARRAY 4591	Group shared data for the specified target must be an array of elements.
ERR_MULTI_SO_NOT_POINT 4592	When multiple geometry shader output streams are used they must be point lists.
ERR_INVALID_SNAP_OFFSET 4593	The target's snap offset must be in the range -8 to 7.
ERR_CLIPPLANE_TOO_COMPLICATED 4594	Clip planes can't be addressed in the specified target; or, clip planes must be non-literal constants with identity swizzles in the specified target.

Constant/value	Description
ERR_ONLY_ONE_ALLOC_CONSUME 4595	RWStructuredBuffer objects can increment or decrement their counters, but not both.
ERR_TYPED_UAV_WRITE_MASK_MISMATCH 4596	Typed UAV stores must write all declared components.
ERR_TEX1D_UNSUPPORTED 4596	Texture1D types are unsupported on the specified target.
ERR_RESINFO_Z_UNDEFINED_CUBEARRAY 4598	The array element count of GetDimensions on TextureCubeArray objects is unavailable on the specified target.
ERR_INVALID_STRUCTURED_ELEMENT_SIZE 4599	The structured buffer element size is invalid. It must be a multiple of specified bytes in the specified target, or it can't be larger than the specified bytes in the specified target.
ERR_MAX_ICB_REG_EXCEEDED 4600	The shader's indexable literal values were exceeded. The shader uses too many indexable literal values so consider using less constant arrays.
ERR_MAX_CBUFFER_SIZE_EXCEEDED 4601	The size of the specified constant buffer is the specified number 16-byte entries, which exceeds maximum allowed size of entries.
ERR_LIB_DEBUG_INST_UNSUPPORTED 4602	Debug instructions are unsupported in shader libraries.
ERR_VARYING_INDEXED_INTERFACE 4603	Interface calls can't be indexed with varying values.
WAR_GEN_NOT_YET_IMPLEMENTED 4700	A feature isn't implemented yet.
WAR_BIAS_MISSED 4701	A _bias opportunity was missed because the source wasn't clamped 0 to 1.
WAR_COMP_MISSED 4702	A complement opportunity was missed because the input result was clamped from 0 to 1.
WAR_LRP_MISSED 4703	Lerp can't be matched because the lerp factor is not _sat'd.

Constant/value	Description
WAR_MAX_CONST_RANGE 4704	Literal values outside range -1 to 1 are clamped on all ps_1_x shading models.
WAR_DEPRECATED_INPUT_SEMANTIC 4705	The specified input semantic has been deprecated; use the specified semantic instead.
WAR_DEPRECATED_OUTPUT_SEMANTIC 4706	The specified output semantic has been deprecated; use the specified semantic instead.
WAR_TEXCOORD_CLAMP 4707	The texcoord inputs used directly (that is, other than sampling from textures) in shader body in ps_1_x are always clamped from 0 to 1.
WAR_MIDLEVEL_VARNOTFOUND 4708	The mid-level var was not found.
WAR_OLD_SEMANTIC 4710	The semantic is no longer in use.
WAR_DUPLICATE_SEMANTIC 4711	A duplicate non-system value semantic definition was encountered.
WAR_CANT_MATCH_LOOP 4712	The loop can't be matched because the loop count isn't from an integer type.
WAR_BIAS_CLAMPED 4713	The sample bias value is limited to the range [-16.00, 15.99] so use the specified value instead of this value.
WAR_CS_TEMP_EXCEEDED 4714	The sum of temp registers and indexable temp registers times the specified number of threads exceeds the recommended total number of threads so performance might be reduced.
WAR_UNWRITTEN_SI_VALUE 4715	A system-interpreted value is emitted that can't be written in every execution path of the shader.
WAR_PSIZE_HAS_NO_SPECIAL_MEANING 4716	The specified semantic has no special meaning on 10_level_9 (9_x feature levels) targets.
WAR_DEPRECATED_FEATURE 4717	Effects are deprecated for the D3DCompiler_47.dll or later.

Requirements

Requirement	Value
Header	CompErrors.h